VTT TIEDOTTEITA – MEDDELANDEN – RESEARCH NOTES 1908

Joustavat ohjelmistoratkaisut tehtäväkriittisessä hajautetussa järjestelmässä

Arno Tuominen
VTT Elektroniikka

VALTION TEKNILLINEN TUTKIMUSKESKUS
ESPOO 1998

ISBN 951-38-5305-5 (nid.)
ISSN 1235-0605 (nid.)

ISBN 951-38-5309-8 (URL: http://www.inf.vtt.fi/pdf)
ISSN 1455-0865 (URL: http://www.inf.vtt.fi/pdf)

JULKAISIJA – UTGIVARE – PUBLISHER

Valtion teknillinen tutkimuskeskus (VTT), Vuorimiehentie 5, PL 2000, 02044 VTT
puh. vaihde (09) 4561, faksi (09) 456 4374

Statens tekniska forskningscentral (VTT), Bergsmansvägen 5, PB 2000, 02044 VTT
tel. växel (09) 4561, fax (09) 456 4374

Technical Research Centre of Finland (VTT), Vuorimiehentie 5, P.O.Box 2000, FIN–02044 VTT, Finland
phone internat. + 358 9 4561, fax + 358 9 456 4374

VTT Elektroniikka, Sulautetut ohjelmistot, Kaitoväylä 1, PL 1100, 90571 OULU
puh. vaihde (08) 509 111, faksi (08) 551 2320

VTT Elektronik, Inbyggd programvara, Kaitoväylä 1, PB 1100, 90571 ULEÅBORG
tel. växel (08) 509 111, fax (08) 551 2320

VTT Electronics, Embedded Software, Kaitoväylä 1, P.O.Box 1100, FIN–90571 OULU, Finland
phone internat. + 358 8 509 111, fax + 358 8 551 2320

Toimitus Leena Ukskoski

Libella Painopalvelu Oy, Espoo 1998

Tuominen, Arno. Joustavat ohjelmistoratkaisut tehtäväkriittisessä hajautetussa järjestelmässä. Espoo 1998, Valtion teknillinen tutkimuskeskus, VTT Tiedotteita – Meddelanden – Research Notes 1908. 74 s.

Avainsanat distributed systems, databases, manufacturing systems, FMS, ECA

Tiivistelmä

Julkaisussa selvitetään ohjelmiston joustavuusominaisuuksien ratkaisu- ja käyttö mahdollisuuksia hajautetussa tehtäväkriittisessä järjestelmässä, jossa on myös reaaliaikaisia ohjelmisto-osia. Joustavuusominaisuuksien ratkaisumallit perustuvat tässä tutkimuksessa CORBA-arkkitehtuuriin ja aktivoituun oliotietokantaan. Tietokannan aktiivisuuspiirteiden toteuttamisessa käytetään ECA-konseptia eli tapahtuma-ehto-toimintomallia. Työ liittyy tutkimusprojektiin, jossa toteutetaan pilottijärjestelmä. Järjestelmän sovellusalueena on tehtäväkriittinen joustava valmistusjärjestelmä (Flexible Manufacturing System), jonka vasteaikavaatimukset ovat vähäiset. Toteutuksessa sovelletaan työnantajan valitsemia konsepteja.

Julkaisun tavoitteena on vastata kahteen päätutkimusongelmaan: mitä ohjelmiston joustavuusmekanismien käytöllä saavutetaan verrattuna komponenttipohjaiseen ohjelmistoarkkitehtuuriin ja miten joustavuusominaisuudet tulee huomioida tehtävä kriittisessä hajautetussa järjestelmässä.

Järjestelmän ohjattavista komponenteista muodostuisi luodulla arkkitehtuurilla perinteisiä monoliittisia sovelluksia, joihin olisi toteutettu kaikki muutettavat tuoteominaisuudet. ECA-säännöillä aktivoitu tietokanta toimisi tuoteominaisuuksien kytkimenä. CORBA-arkkitehtuurin mahdollistama joustava laajennettavuus tuo lisäarvoa järjestelmälle, mutta järjestelmän hajauttaminen itsenäisesti toimiviin olioihin aiheuttaa ongelmia järjestelmän ohjauksessa.

Joustavuuden saavuttamiseksi tulee tuntea ohjattava järjestelmä ja sen varioitavat tuoteominaisuudet. ECA-sääntöjen muodostamiseen ei pilottijärjestelmässä tehdyn työn pohjalta voi esittää yhtä ainoaa oikeaa vaihtoehtoa, vaan säännöt muodostuvat ohjattavan toiminnan mukaan. ECA-konseptin staattinen CORBA-toteutus rajoittaa joustavuutta ja järjestelmän onnistunut luokkahierarkia muodostuu tärkeäksi tekijäksi joustavuuteen pyrittäessä.

Alkulause

Tutkimuksen lähtökohtana oli Valtion teknillisen tutkimuskeskuksen VTT Elektroniikan tutkimusyksikön tutkimusprojekti. Projektissa tehtävä pilottiohjelmisto sekä siitä saatava kokemus toimivat tutkielman perustana. Tarkoituksena on lisätä hajautettujen järjestelmien, oliotietokantojen ja aktiivisten tietokantojen tietämystä. Olennaisena osana työhön liittyy myös suomalaiselle teollisuudelle suoritettava teknologian siirto, jolla pyritään parantamaan kansallista kilpailukykyä ja helpottamaan yritysten päätöksentekoa uusista teknologiastrategioista.

Parhaimmat kiitokset työni ohjauksesta haluan osoittaa tutkielman ohjaajina toimineille fil.tri Ilkka Tervoselle ja fil. maist. Eila Niemelälle. Suuren kiitoksen ansaitsee Mikko Holappa ystävällisestä opastuksesta CORBA-maailman saloihin ja epäitsekkäästä toiminnasta samassa tutkimusprojektissa. Erityisen kiitoksen ansaitsee myös Jani Granholm Oy Mercantile AB Fastems:lta merkittävästä panoksestaan joustavien valmistusjärjestelmien sovellusalueeseen perehdyttämisessä.

Lisäksi haluan muistaa kiitoksella opintojeni varrella saamastani tuesta vanhempiani sekä Saara Björkbackaa ja hänen perhettään. Opiskeluajan yhteisistä työhetkistä haluan kiittää Esa Tikkalaa ja Tomi Leppikangasta.

Tämä tutkielma päättää yli 20 vuotta kestäneen koulutieni — hetkeksi.

Oulussa 27.4.1998

Arno Tuominen

Sisällysluettelo

Lyhenteet

ADBMS	Active Database Management System
API	Application Programming Interface
ATM	Agent Transfer Manager
CORBA	Common Object Request Broker Architechture
DCOM	Distributed Component Object Model
DII	Dynamic Invocation Interface
DML	Data Manipulation Language
ECA	Event-Condition-Action
FMS	Flexible Manufacturing System
IDL	Interface Definition Language
IIOP	Internet Inter-ORB Protocol
MFC	Microsoft Foundation Classes
MOM	Message Oriented Middleware
ODA	Object Database Adaptor
ODBMS	Object Database Management System
ODMG	Object Database Management Group
OMG	Object Management Group
ORB	Object Request Broker
OTS	Object Transaction Service

POS	Persistent Object Service
RPC	Remote Procedure Call
URL	Uniform Resource Locator

1. Johdanto

Tutkielman tavoitteena on selvittää ohjelmiston joustavuusominaisuuksien ratkaisu- ja käyttömahdollisuuksia hajautetussa pehmeässä reaaliaikajärjestelmässä. Joustavuusominaisuuksien ratkaisumallit perustuvat tässä tutkimuksessa CORBA-arkkitehtuuriin ja aktiiviseen olio tietokantaan. Tietokannan aktiivisuuspiirteiden toteuttamisessa käytetään ECA-konseptia eli tapahtuma-ehto-toimintomallia. Työnantaja on valinnut toteutukseen käytettävät konseptit.

1.1 Taustatietoa tutkimuksesta

Joustavuusominaisuudet liittyvät olennaisesti ohjelmiston tuotteistamiseen. Ohjelmointiprojektin sijasta olisi järkevää pystyä lyhyelläkin aikavälillä myymään tuote eikä yksittäistä projektia. Komponentoimalla ohjelmisto ja kehittämällä mekanismit joustavuuden saavuttamiseksi on mahdollista saavuttaa suuria säästöjä. Ohjelmiston laatu ja asiakastyytyväisyys paranevat ja toimitukset lisäksi nopeutuvat. Ohjelmistojen hallinta ja after-sales-palvelujen tuottaminen helpottuvat. Aihealueeseen liittyvät käsitteinä myös ohjelmiston ja sen osien uudelleenkäyttö ja ohjelmistotuotteen hallinta.

Tämä tutkimus tehtiin VTT Elektroniikassa TEKESin tavoitetutkimusprojektiin liittyvänä opinnäytetyönä. Tutkimusprojektiin liittyen toteutettiin pilottijärjestelmä, jonka sovellusalueena on joustava valmistusjärjestelmä (Flexible Manufacturing System). Järjestelmä on tehtäväkriittinen — jokainen tuotantovaihe oli saatettava loppuun ennen seuraavan aloittamista.

Käytettäessä koneiden ohjausjärjestelmiä useita vuosia samalla tuotealueella muodostuu järjestelmätuotteista usein ryhmiä. Näitä ryhmiä joudutaan jatkuvasti muuttamaan kaupallisten tai asiakkaalla käytössä olevien teknologioiden takia sekä myös asiakkaalta tulevista suorista vaatimuksista johtuen. Tuoteperheelle saadaan aikaan mukautumiskykyä suunnittelemalla geneerinen arkkitehtuuri ja konfiguroitavat kommunikointimekanismit sekä liityntäluokat järjestelmän eri komponenttien välille. Adaptiivisen ohjelmiston haittoja ovat lisääntynyt muistin käyttö, pidentyneet vasteajat ja lisääntynyt kehitystyön resurssien tarve. (Ihme & Niemelä 1996, s. 29, 36.)

Reaaliaikajärjestelmät jaetaan ympäristön asettamien aikarajoitteiden mukaan joko pehmeisiin (soft) tai koviin (hard) reaaliaikajärjestelmiin. Pehmeille reaaliaikajärjestelmille sallitaan satunnainen vasteaikojen ylittyminen. Kovassa reaaliaikajärjestelmässä jopa yksittäisen vasteaikavaatimuksen rikkominen johtaa virheeseen. (Päivike 1991, s. 1.)

Vasteaikavaatimukset ovat kehitetyssä pilottijärjestelmässä kuitenkin pehmeät, mikä antaa mahdollisuudet soveltaa järjestelmässä ‘hitaitakin’ komponentteja, kuten oliotietokantaa. Joustavuutta pyritään saavuttamaan ECA-säännöillä ja aktivoidulla oliotietokannalla, johon talletetaan tapahtumien pohjalta suoritettavat toiminto-ohjeet laitteiston ohjaamiseksi. CORBA-

arkkitehtuurin käytöllä pyritään saavuttamaan viestinvälityksen joustavuus hajautetun valmistusjärjestelmän eri osien välille.

Oliotietokanta on luonteva ratkaisu aktiivisen tietokannan toteuttamiseen tässä ympäristössä, koska toteutettu pilottiohjelmisto perustuu olioteknologian käyttöön. Tietokanta muodostaa perustan, johon järjestelmän toimintaa ohjaavat säännöt tallennetaan. Tietokannan valintaa edelsi VTT Elektroniikan suorittama teknologiaselvitys, jossa kartoitettiin eri tietokantavaihtoehtoja. Tietokannaksi valittiin Objectivity Inc:n Objectivity/DB 4.0.2 -oliotietokanta. Valintaa puolsivat hyvä hinnan suhde ominaisuuksiin ja tulossa oleva sovitin IONA Technologiesin Orbix ORB-toteutukseen. Orbix oli selvä valinta ORB-toteutukseksi johtuen luvatusta sovittimesta, ja lisäksi tuotteen aiemmasta versiosta oli kertynyt käyttökokemusta, käytetyn tuotteen versio on 2.2C01 MT. Käyttöjärjestelmäksi valittiin Microsoft Windows NT 4.0 paljolti markkinavoimista johtuen.

1.2 Tutkimusongelmat ja metodi

Tutkielman tavoitteena on vastata seuraaviin tutkimusongelmiin:

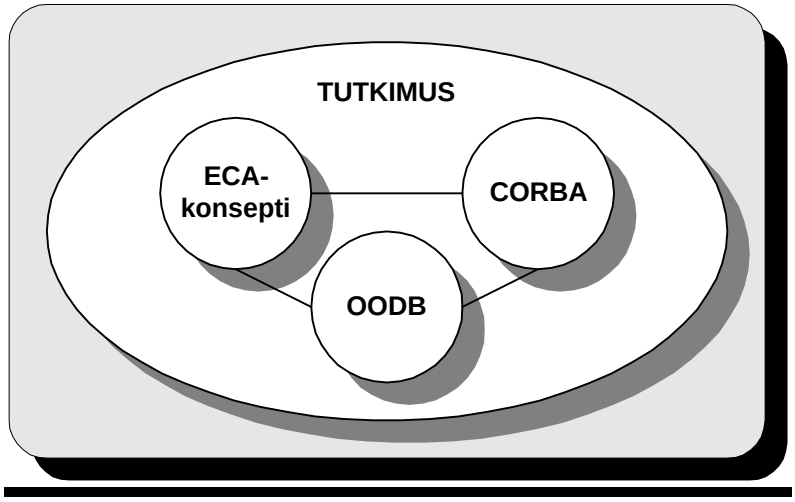
1. Mitä ohjelmiston joustavuusmekanismien käytöllä saavutetaan verrattuna komponenttipohjaiseen ohjelmistoarkkitehtuuriin?

2. Miten joustavuusominaisuudet tulee huomioida tehtäväkriittisessä hajautetussa järjestelmässä?

Tutkimus on luonteeltaan konstruktiivinen soveltava tutkimus. Tässä tutkimusmallissa tavoiteltava lopputila on tiedossa etukäteen, mutta menetelmiä tuloksen saavuttamiseksi ei tiedetä (Järvinen & Järvinen 1995, s. 72). Tutkimuksen teoriaosassa pyritään muodostamaan yleinen kuva tutkielmaan liityvistä aihealueista ja käytetyistä konsepteista, joilla joustavuuteen pyritään. Tutkimusongelmiin vastataan projektiin liittyvän pilottijärjestemän toteutuksessa kertyneisiin kokemuksiin ja kirjallisuudesta saatavilla olevaan tietoon pohjautuen.

Tämän tutkimuksen ulkopuolelle rajataan muut hajautuksen mahdollistavat asiakas-palvelin-arkkitehtuurit, kuten DCOM ja Java. Niin ikään ulkopuolelle rajataan hajautukseen liittyvien yksityiskohtien ja vaatimusten tarkempi tarkastelu ja laitteistoihin liittyvät asiat. Kuvassa 1 on tutkimuksen rajaus esitettynä graafisesti. Tutkimus keskittyy ECA-mallin sovittamiseen kaupalliseen oliotietokantaan ja aikaansaadun kombinaation yhdistämiseen CORBA-arkkitehtuuriin.

Lisäksi haetaan vastauksia seuraaviin kysymyksiin. Mitä ECA-konseptin soveltamisella voidaan saavuttaa hajautetussa tuotannonohjausjärjestelmässä? Miten CORBA-arkkitehtuurin käyttäminen vaikuttaa aktiivisuuspiirteiden toteuttamiseen tietokannassa? Mitkä ovat ratkaisuvaihtoehdot joustavuuden saavuttamiseksi ja miten ne tulee huomioida suunnittelussa?



Kuva 1. Tutkimuksen rajaus.

Tutkielman työvaiheet muodostuivat valmistusjärjestelmien sovellusalueeseen ja sen terminologiaan, ECA-konseptiin ja käytettyyn oliotietokantaan, sekä CORBA-arkkitehtuuriin ja sen toteuttavan ORB-teknologiaan perehtymisestä. Aluksi ECA-konseptista muodostettiin muistinvarainen luokkamalli käyttäen apuna MFC-luokkakirjastoa. Tämä luokkamalli yhdistettiin CORBA-arkkitehtuuriin ja toimintaa testattiin ilman oliotietokantaa. Lopuksi luokat tallennettiin oliotietokantaan ja luokkamallia laajennettiin, jotta liittyminen ohjattavan järjestelmän osiin saatiin aikaan. Järjestelmän ohjaukseen tehtiin käyttöliittymät soveltaen luotua arkkitehtuuria.

1.3 Tutkimuksen rakenne ja tutkimusaineisto

Tutkielma jakaantuu kahteen osa-alueeseen: teoriaosaan, joka muodostuu luvuista 2 ja 3, sekä luvun 4 konstruktiiviseen osaan. Teoriaosassa käytetyt lähteet jakautuvat kolmeen eri osa-alueita käsittelevään ryhmään, CORBA-arkkitehtuuria, ECA-konseptia sekä luvun kaksi taustatietoa käsitteleviin lähteisiin. Käytetyimmät CORBA-arkkitehtuuria käsittelevät lähteet ovat Bakerin (1997) CORBA Distributed Objects Using Orbix ja Orfalin (et al. 1997) Instant CORBA. Tietokantoja käsittelevistä lähteistä tärkeimmät ovat Barryn (1996) The Object database handbook: how to... sekä Elmasrin ja Navathen (1994) Fundamentals of database systems. Tärkein yksittäinen artikkeli on Dittrichin (et al. 1995) The Active Database Management System Manifesto: A Rulebase of ADBMS Features.

Toisessa luvussa käsitellään olennaisimpia aiheeseen liittyviä käsitteitä kirjallisuuteen pohjautuen. Kolmas luku esittelee tarkemmin kaksi tutkielmassa sovellettua konseptia, joilla joustavuutta pyritään saavuttamaan. Ensiksi käsitellään oliotietokantoja ja aktiivisuutta oliotietokannoissa yleisesti. Tämän jälkeen käydään läpi aktiiviselle tietokannalle esitettyjä vaatimuksia ja esitellään itse ECA-konsepti. Toisena osa-alueena kolmannessa luvussa käsitellään tarkemmin CORBA-

arkkitehtuuria ja sen tarjoamia palveluja eri laitteistoarkkitehtuurien kommunikointiin ja keskitytään kahteen palveluun. Nämä palvelut ovat pysyvyyspalvelu (POS) ja transaktiopalvelu (OTS). Lopuksi käsitellään oliotietokantojen yhdistämistä CORBA-arkkitehtuuriin.

Neljäs luku muodostuu pilottijärjestelmän toteutuksesta saatavasta tiedosta. Luvussa esitellään projektissa tehtyä työtä; sitä, kuinka arkkitehtuurimallit yhdistettiin ja miten liittyminen ohjattavaan järjestelmään tapahtuu. ECA-konseptin toteutusta ja siinä esiintyneitä ongelmia käsitellään tarkemmin. Lopuksi kerrotaan, miten arkkitehtuuria voidaan käyttää pilottijärjestelmän toiminnallisuuden muuttamiseen sekä kuinka käyttöliittymä yhdistettiin ECA-konseptiin.

Viidennessä luvussa analysoidaan tuloksia ja esitetään vaihtoehtoja järjestelmän kehittämiseksi. Pilottijärjestelmässä aktivoitua tietokantaa verrataan Dittrichin (et al. 1995) esittämiin aktiiviselle tietokannalle asetettuihin vaatimuksiin. Lopuksi kuudennessa luvussa esitetään yhteenveto tehdystä työstä ja vastataan tutkimusongelmiin.

2. Määritelmiä ja käsitteitä

Tässä luvussa tarkastellaan lyhyesti tutkielman aihealueeseen liittyviä määritelmiä ja käsitteitä. Aluksi käsitellään joustavia valmistusjärjestelmiä ja komponentointia. Kirjallisuudessa esitettyjä määritelmiä agenteille ja välitason ohjelmistolle käsitellään lyhyesti. Lopuksi perehdytään oliotietokantoihin.

2.1 Joustava valmistusjärjestelmä

Joustavalla valmistusjärjestelmällä (flexible manufacturing system) tarkoitetaan Mortimerin (1982, s. 9) mukaan kontrolloitua prosessia, joka tuottaa useita erilaisia komponentteja tai tuotteita ennalta määrätyn aikataulun ja kapasiteetin rajoissa. Mortimer (1982) esittää myös toisen määritelmän, jossa FM-järjestelmä nähdään toteuttavana teknologiana, joka auttaa rakentamaan kilpailukykyisempiä tehtaita nopeammilla vasteajoilla sekä saavuttamaan alhaisemmat yksikkökustannukset, paremman laadun ja kehittyneemmän tuotannonhallinnan ja pääoman kontrolloinnin.

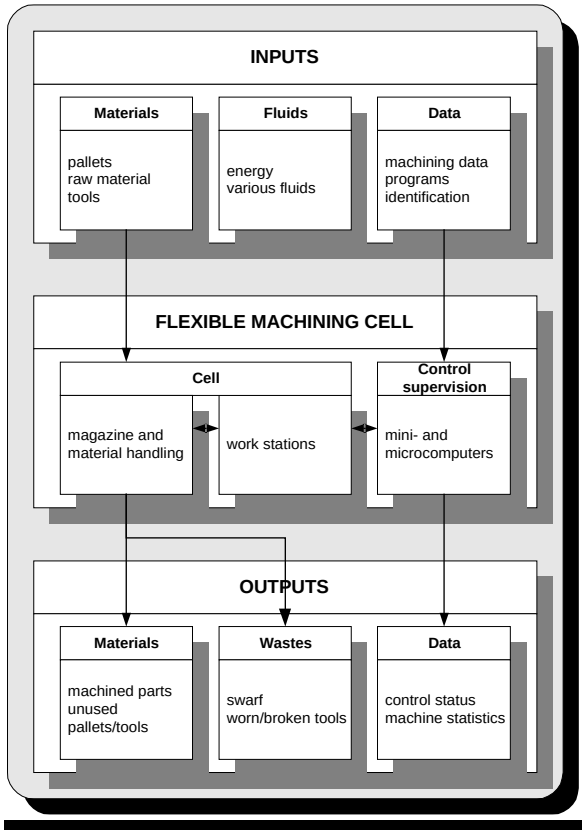
Hartleyn (1984) mukaan FM-järjestelmän keksijänä pidetään englantilaista Theo Williamsonia. Hän kehitti vuonna 1968 konseptin, jota kutsuttiin nimellä ‘System 24’, koska järjestelmä oli suunniteltu toimimaan vuorokauden ympäri. (Greenwood 1986, Hartley 1984). Williamson suunnitteli numeerisesti ohjattavia työstökoneita käytettäväksi työstettävien kappaleiden koneistusoperaatioissa. Työstettävät kappaleet asennettaisiin käsin kuljetusalustoilleen, jotka voitaisiin automaattisesti siirtää työstökoneille. Jokaiseen koneeseen kuuluisi työkalumakasiini, josta automaattisesti ladattaisiin eri työstövaiheissa tarvittavat työkalut. Konseptiin kuului myös jäysteiden poisto ja työstettyjen kappaleiden pesu. (Hartley 1984, s. 2.)

Williamsonin järjestelmä yhdisti tietokoneohjattujen järjestelmien joustavuuden ja vähäisen työntekijöiden sitomisen järjestelmän toimintaan. Robottien ja muiden tietokoneohjattujen laitteiden myötä FMS-konseptia voidaan soveltaa myös muihin prosesseihin kokoonpanotehtävistä aina metallin työstämiseen. Ensimmäiset asennetut FM-järjestelmät noudattivat Williamsonin luomaa konseptia laajentaen sitä työkalujen kulumisen ja rikkoutumisen seurannalla. (Hartley 1984, s. 2.) FM-järjestelmän peruselementit ja toiminnallisuus näkyvät kuvassa 2.

Greenwood (1986, s. 3) luettelee kaikille FM-järjestelmille tyypilliset komponentit, jotka ovat

- ohjelmistosta ja laitteistosta koostuva ohjausjärjestelmä
- viestintäjärjestelmä, joka muodostuu ohjelmistosta, tiedonvälitykseen käytettävästä mediasta, sekä laitteistosta, jota tarvitaan tietokone- ja prosessilaitteisiin liittymiseen,
- prosessissa käytettävät laitteet, kuten robotit, työstökoneet ja numeerisesti ohjattavat laitteet.

Näistä tärkeimpinä Greenwood (1986) pitää kahta ensin mainittua, koska ne määräävät järjestelmän menestymisen tai epäonnistumisen, niin teknisesti kuin kaupallisestikin.



Kuva 2. FM-järjestelmän yleinen toiminnallisuus (Hartley 1984, s. 4).

2.2 Komponentointi

Derin (1997) mukaan komponentti on uudelleenkäytettävissä oleva osa ohjelmistoa, joka voidaan suhteellisen helposti liittää muiden valmistajien tuottamiin komponentteihin. Vastakohtana komponentille on perinteinen monoliittinen sovellus eli ohjelmisto, johon on ennalta sisällytetty suuri määrä valittuja ominaisuuksia. Näistä ominaisuuksista suurinta osaa ei voi korvata tai poistaa (Deri 1997). Derin (1997) kehittämä komponenttipohjainen Yasmin-arkkitehtuuri kehitettiin painottaen tietoverkon hallintaan tarkoitettujen hajautettujen sovellusten toteuttamisen yksinkertaistamista. Ideana on se, että käyttäjä voi koota haluamansa sovelluksen komponenteista ja poistaa tai vaihtaa niitä ajon aikana.

Kalaoja et al. (1997) ehdottavat kerrostettujen tuoteominaisuusmallien käyttämistä komponenttipohjaisen ohjelmiston kehittämiseen ja tuotetiedon uudelleenkäyttöön. Tuoteominaisuusmalli määrittelee tuoteperheen kaikki mahdolliset ominaisuudet sekä niiden keskinäiset riippuvuussuhteet. Ominaisuusmalliin perustuen komponenttipohjaisen suunnittelun tavoitteena on

sellaisten parametroitavien ohjelmakomponenttien määritteleminen, jotka toteuttavat kaikki tuote ominaisuusmallin piirteet. Lopulliseen toimitettavaan tuotteeseen sisällytetään ainoastaan vaaditut ominaisuudet toteuttavat komponentit. Laatu ja tehokkuus parantuvat epäonnistuneiden toimitusten määrän vähentyessä sekä komponenttien uudelleenkäytön myötä. Tuoteominaisuusmallin tekeminen vaatii kokemusta sovellusalueesta ja ohjelmistoarkkitehtuurin, joka tukee komponentoinnilla ja parametreilla aikaansaatavaa joustavuutta. Lisäksi tarvitaan mekanismi, joka liittää ohjelmakomponentit ja tuoteominaisuudet toisiinsa. Yleistä ratkaisua esitettyihin vaatimuksiin ei ole olemassa, vaan ratkaisut ovat yritys- ja tuotekohtaisia. (Kalaoja et al. 1997)

Tranin (et al. 1997) mukaan pääasiallisia syitä komponenttipohjaisen ohjelmistokehityksen kiinnostuksen kasvuun ovat mm.

- lisääntyvä teollinen kilpailu yhä laadukkaampien ja luotettavampien tuotteiden aikaansaamiseksi yhä lyhyemmässä ajassa
- yhä suurempien ja monimutkaisempien ohjelmistojen lisääntyvä tarve, joita ei voida enää toteuttaa asetetuissa aikarajoissa yhden organisaation toimesta
- uudelleenkäytettävien kaupallisten ohjelmistokomponenttien lisääntynyt saatavuus
- tuotteiden yhdistämiseen käytetyn ajan vähentyminen johtuen valmiiden kaupallisten komponenttien parantuneesta yhteensopivuudesta ja standardien noudattamisesta
- komponentointitekniikoiden ja lähestymistapojen lisääntynyt tutkimus ja tuotekehitystuki
- yhä lisääntynyt uudelleenkäytön arvostus yhtenä tärkeimmistä keinoista parempien ratkaisujen kehittämiseksi mahdollisimman vähäisin kehityskustannuksin.

Valmiiden komponenttien valintaan, yhdistämiseen ja evaluointiin liittyy kuitenkin teknisiä riskejä, joita ei tule aliarvioida (Tran et al. 1997). Komponenttien välisten riippuvuussuhteiden tunnistaminen ja hallinta muodostuu uuden paradigman suurimmaksi huolenaiheeksi; uusien komponenttipohjaiseen suunnitteluun tarkoitettujen metodologioiden tulisikin korostaa tätä seikkaa omana suunnitteluongelmanaan (Dellarocas,1997).

2.3 Agentit

Woolridge ja Jennings (1995) esittävät agenteille kaksi määritelmää, vahvan ja heikon. Agentin heikko määritelmä ilmaisee laitteisto- tai ohjelmistopohjaista järjestelmää jolla on seuraavat ominaisuudet:

- autonomisuus; agentit toimivat ilman, että kukaan ihminen tai muu tekijä puuttuu niiden toimintaan. Agenteilla on lisäksi jonkinlainen kontrolli sisäisestä tilastaan ja toiminnoistaan
- yhteistyökyky; agentit toimivat yhdessä ihmisten tai toisten agenttien kanssa jollain tehtävään tarkoitetulla kielellä (Genesereth & Ketchpel 1994, ks. Woolridge & Jennings 1995)

- reaktiivisuus; agentit havainnoivat ympäristöään ja vastaavat siinä tapahtuviin muutoksiin ajoitetusti
- ennakoiva käyttäytyminen; agentit voivat tehdä aloitteen toiminnallisuuteen eivätkä pelkästään tuottaa vasteita ympästöstä tulleisiin herätteisiin pyrkien tiettyyn päämäärään.

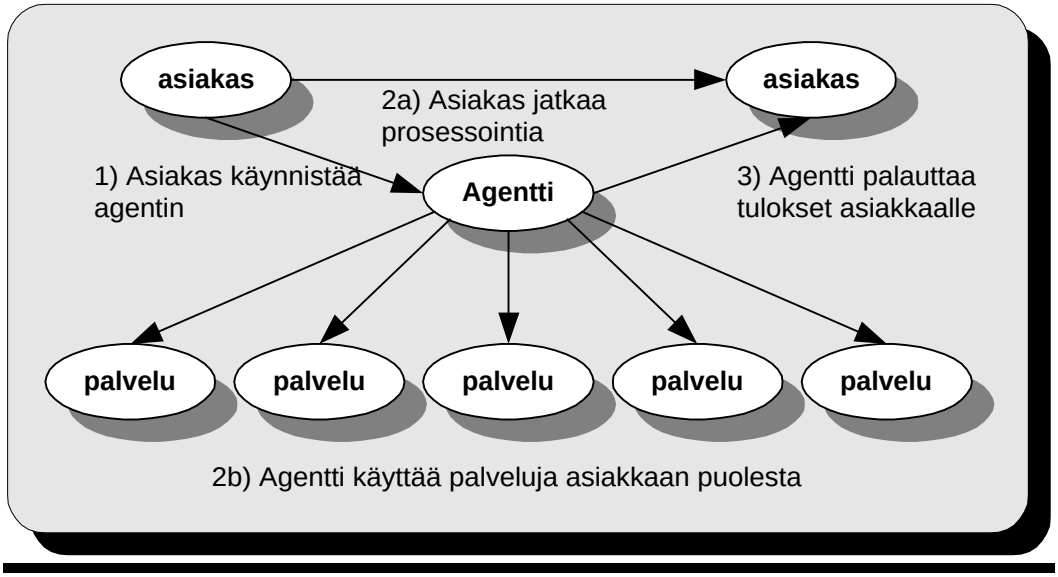
Tekoälytieteissä agentilla on tarkempi ja vahvempi merkitys. Silloin agentti yleisesti tarkoittaa järjestelmää, jolla on edellä mainittujen ominaisuuksien lisäksi toteutettuna toiminnallisuutta, joka yhdistetään ihmisen käyttäytymiseen. (Woolridge & Jennings 1995, s. 2.)

Orfali et al. (1997, s. 268) määrittelevät liikkuvan agentin CORBA-olioksi, joka voi siirtää koodinsa ja tilansa IIOP-verkossa paikasta toiseen ja poikkeaa näin Java-appletista tai -beanista. Agentti aloittaa toimintansa itsenäisesti sen saapuessa määränpäähänsä, mutta toimii kuitenkin AT-managerin (agent transfer manager) alaisuudessa, joka vaaditaan jokaiseen agentin vastaanottavaan koneeseen. Liikkuvalla agentilla on tunnisteensa (URL) ja tietty kulkureitti, jota se noudattaa. Agentin vastaanottavaa palvelinta suojataan suorittamalla agenttia tietyn kontekstin alaisuudessa. Agentti-palvelu on spesifikaationa kuitenkin vielä keskeneräinen ja se on todennäköisesti tulossa CORBA-arkkitehtuurin yleisiin lisäpalveluihin (common facilities) CORBA 3.0 -spesifikaatiossa. Tämäkin palvelu on muiden palvelujen tapaan määritelty IDL-kielellä. (Orfali et al. 1997.)

Mowbray & Malveau (1997) esittävät suunnittelumallin agenttien soveltamiseksi, jonka tarkoituksena on erilaisten informaatiopalvelujen käytön yksinkertaistaminen ja yhtenäistäminen asiakkaan kannalta. Malli on käyttökelpoinen silloin, kun tarvitaan tehtävien delegointia muille sovelluksille tai asiakas tarvitsee useita eri palveluja. Lisäksi mallia tulisi käyttää,

- kun asiakas tarvitsee koottuja tuloksia
- kun eri tehtävien toiminnan ohjaaminen on monimutkainen tehtävä
- kun asiakas tarvitsee ilmoituksen jonkin tehtävän suorituksesta.

Agentin kehittämiseen tarvitaan kolme vaihetta. Ensiksi tulee päättää, mitä toimintoja delegoidaan agentille. Toiseksi tulee sopia, kuinka agentti initialilisoidaan, ja lopuksi tarvitaan mekanismi tulosten siirtämiseksi asiakkaalle. Etuina saavutetaan järjestelmään lisää rinnakkaisuutta eri tehtävien välille ja samalla mahdollistetaan asiakkaan toiminnan jatkuminen agentin suorittaessa tehtäviä. Varjopuolena agentti vaatii oman prosessin ja välttämättä mitään vastausta ei koskaan saada vakavan virheen sattuessa. (Mowbray & Malveau 1997, s. 202 - 203.) Mallin toiminta on kuvassa 3.



Kuva 3. Agentti-mallin toiminta (Mowbray & Malveau 1997, s. 202).

2.4 Välitason ohjelma

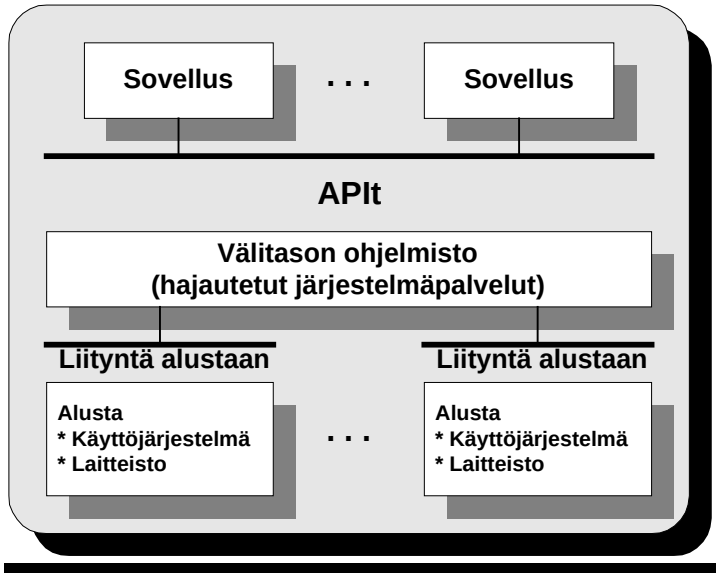
Mowbrayn ja Zahavin (1995, s. 235) mukaan termiä välitason ohjelma (middleware) käytetään kuvaamaan laajaa, kaupallisten järjestelmien integrointiin tarkoitettujen ohjelmien joukkoa. Termi jakaatuu kahteen pääkategoriaan,

- hajautettuun prosessointiin tarkoitettuun välitason ohjelmistoon (alempaan välitasoon)
- käyttöliittymiin ja tietokantohin tarkoitettuun välitason ohjelmistoon (ylempiään välitasoon).

Bernstein (1996) määrittelee välitason ohjelmistopalvelun kuvan 4 mukaisesti yleiskäyttöiseksi palveluksi, joka on laitteistoalustojen ja laitteistoja käyttävien sovellusten välissä. Sovelluksen kannalta välitason ohjelmisto tekee tietoverkolle saman, minkä käyttöjärjestelmä tekee tietokonelaitteistolle (King 1992, s. 59). Tarkasteltaessa usean protokollan ja alustan järjestelmää myös etäkutsut (RPC) ja SQL-kyselyt voidaan lukea välitason ohjelmistoiksi, koska ne mahdollistavat tiedon vaihtamisen, tallentamisen ja palauttamisen piilottaen alla olevan tietoverkon monimutkaisuuden. Mikäli hajautettuun tiedonsiirtoon eri alustojen välille tarkoitettu ohjelmointirajapinta lisää ylemmän tason liitynnän, niin sitä voidaan pitää välitason ohjelmistona. (King 1992, s. 59.)

King (1992, s. 62) jakaa välitason ohjelmistot tarkemmin neljään perusluokkaan, jotka ovat: hajautetut tiedostot, funktiokutsut (RPC) ja tietokannat (SQL) sekä hajautettu sanoman välitys. Välitason ohjelmistot voivat kuulua yhteen tai useampaan luokkaan. Tarkemmat kriteerit välitason ohjelmistolle ovat Kingin (1992, s. 62) mukaan alustariippumattomuus, tiedon läpinäkyvyys, tiedon läpinäkyvä muuntaminen, useat yhteysmuodot sovellusten välillä, lisäarvoa tuottavat palvelut ja tietoverkon hallinta sekä vasteet tapahtuneisiin virhetilanteisiin.

Bernsteinin (1996, s. 91) välitason ohjelmistolla tuotetut palvelut eivät kuitenkaan ratkaise kaikkia ongelmia. Useat suositut välitason ohjelmistopalvelut käyttävät valmistajakohtaisia dokumen toimattomia protokollia ja sovellusohjelmaliityntöjä, mikä sitoo ohjelmistot tietyn valmistajan tuotteeseen ja vaikeuttaa keskenään toimivien sovellusten rakentamista. Välitason ohjelmistojen pienimuotoinenkin käyttö voi johtaa ohjelmiston kompleksisuuden suureen kasvuun, ja ohjelmistojen kehittäjät joutuvat valitsemaan tarpeitaan vastaavan pienen määrän palveluja. Välitason ohjelmiston käyttäminen nostaa hajautettujen sovellusten ohjelmoinnin abstraktiotasoa mutta jättää silti ohjelmiston kehittäjälle vaikeita päätöksiä toiminnallisuuden jakamisesta ja funktioiden sijoittamisesta hajautettuihin ohjelman osiin. (Bernstein 1996, s. 91.)



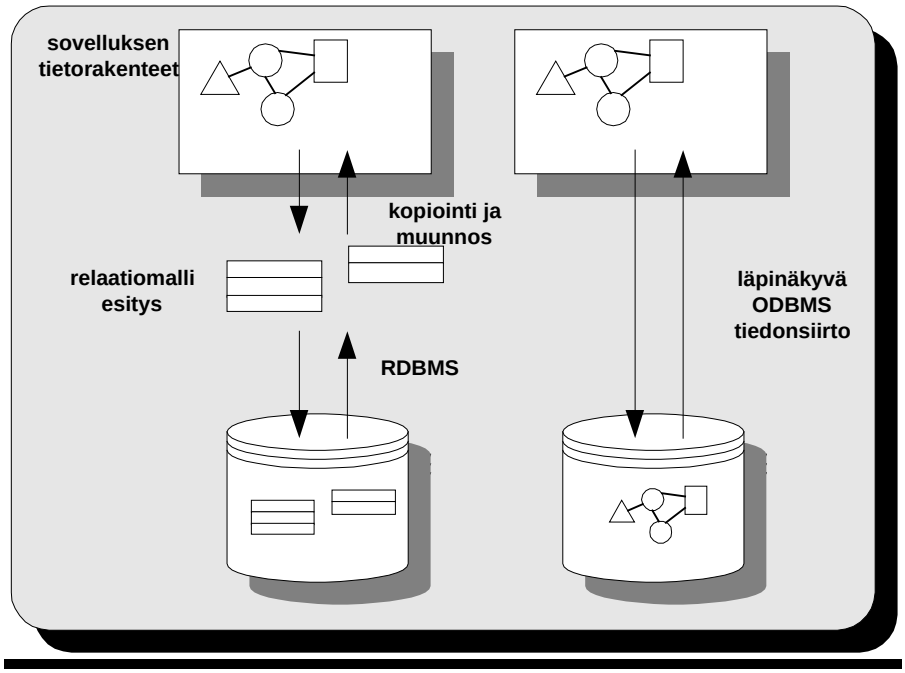
Kuva 4. Välitason ohjelmiston komponentit (Bernstein 1996, s. 89).

Sanomasuuntautuneella välitason ohjelmistolla (MOM) tarkoitetaan ohjelmistoa, jonka avulla palvelimet ja asiakkaat voivat toimia itse haluamanaan aikana ja eri nopeuksin olematta kuitenkaan yhtä aikaa aktiivisina. Toiminta perustuu sekä asiakkaan että palvelimen asynkronisiin sanomajonoihin. (Mowbray & Malveau 1997, s. 249.) Hajautetun prosessoinnin väliohjelmisto on siirtymässä CORBA-arkkitehtuurin käyttöön (Mowbray & Zahavi, 1995).

2.5 Perustietoa oliotietokannoista

Cattell ja Barry (1997) määrittelevät oliotietokannan tietokannaksi, joka yhdistää tietokannan ominaisuudet oliosuuntautuneen ohjelmointikielen ominaisuuksiin. Oliotietokanta saa tietokantaan talletetut oliot näyttämään käytetyn ohjelmointikielen olioilta, koska se käyttää joko yhtä tai useampaa olemassa olevaa ohjelmointikieltä. Oliotietokanta myös laajentaa käytettyä ohjelmointikieltä läpinäkyvällä tiedon pysyvyydellä, rinnakkaisuuden hallinnalla, tiedon palauttamisella, luokkien välisiä kytkentöjä seuraavilla tietokantakyselyillä ja muilla tietokannan ominaisuuksilla. Läpinäkyvyys tekee tarpeettomaksi opetella erillistä kieltä datan käsittelyyn sekä

poistaa datan eksplisiittisen muuttamisen ja kopioinnin tietokannan ja käytetyn ohjelmointikielen esitystavan välillä. Kuvassa 5 on esitettynä arkkitehtuurien ero. (Cattell & Barry 1997, s. 2 - 3.)



Kuva 5. Oliotietokannan ja relaatiotietokannan arkkitehtuurien ero (Cattell & Barry 1997, s. 3).

Kysymykseen kumpi arkkitehtuurimalli on parempi, ei Barryn (1996) mukaan ole yhtä oikeaa vastausta, vaan vastaus riippuu monista tekijöistä. Monimutkainen data on yleisesti hallittavissa parhaiten oliotietokannalla. Monimutkaisen datan tunnusmerkkejä ovat luonnollisen uniikin tunnuksen puuttuminen, paljon monesta-moneen-relaatioita, käsittelyn tarve navigointiin relaatiosta toiseen (use of traversals) ja tyyppikoodien toistuva käyttäminen. (Barry 1996, s. 45 - 47.) Kehittyneet tekniset sovellukset, kuten tietokoneavusteinen suunnittelu ja valmistus, laajentavat perinteisille tietokantasovelluksille asetettuja vaatimuksia. Näitä vaatimuksia ovat esimerkiksi monimutkainen oliomallintaminen sekä tuki versioinnille ja pitkille transaktioille. (Huemer et al. 1995, s. 1065.) Loomisin (1995, s. 12 - 13) mukaan oliotietokantojen käyttöön päädytään seuraavista yleisistä syistä johtuen:

- sovelluskehittäjien päätös käyttää olioteknologiaa; oliotietokannat tulevat mukaan osana tuota ratkaisua

- kehitettävien tietokantasovellusten mutkikkuus; niitä ei voi hallita enää hyvin relaatiomallin avulla

- hajautettujen sovellusten kehittäminen; sovellukset voivat hyötyä oliotietokantojen hajautus- ja työryhmäominaisuuksista.

Taulukossa 1 vertaillaan sitä, kuinka oliomallin käsitteet eroavat toteutuksiltaan relaatiomallissa. Relaatiomallissa ‘Monen suhde moneen’ -relaation esittäminen vaatii välitaulun (liityntäentiteetin) käyttöä ja ohjelmakoodissa eri entiteetit erotetaan toisistaan tyyppikoodin avulla haarautuen. Oliomallissa voidaan viitata suoraan oliosta toiseen käyttäen olioiden tunnisteita. Käytettäessä perintää ei tarvita ylimääräisiä tyyppikoodeja ja oliomallissa on sisäänrakennettu haarautuminen oikeaan koodiin ohjelmaa suoritettaessa.(Barry 1996, s. 49 - 63.)

Taulukko 1. Relaatio- ja oliomallin vertailu (Barry 1996, 62).

Oliomallin käsite	***Relaatiomallin tekniikka***	***Oliomallin tekniikka***	***Oliomallin edut***
datan erottaminen	liityntäentiteetit ja indeksointi viittausten esittämiseen rivien (tuple) välillä	OID:t esittävät suoraan viittauksia olioiden välillä	monimutkaisen datan yksinkertaisempi malli
perintä	tyyppikoodit	luokkahierarkia	suora esitys tyypin ja alityypin suhteesta, tuki jokaiselle tyypille ominaisesta prosessoinnista
kapselointi	tyyppikoodeihin perustuva “If then else if” -ohjelmakoodi	sisäänrakennettu haarautuminen oikeaan koodiin	sovelluskoodin ja virhemahdollisuuden pieneneminen väärän koodin suorittamisesta oikealle datalle

Loftus et al. (1995, s. 139 - 140) jakavat oliotietokanta-arkkitehtuurit neljään kategoriaan seuraavasti:

- Laajennettu relaatiomalli (extended relational). Relaatiomallia on laajennettu oliosuuntautuneilla piirteillä, kuten relaation luokittelevilla hierarkioilla tai metodeilla. Tämän mallin etuna on oliotietokannan rakentuminen olemassa olevalle kypsälle teknologialle, joka tukee rinnakkaisuutta ja transaktioiden hallintaa sekä relaatioihin tehtäviä kyselyjä. Olemassa olevan tekniikan laajentaminen johtaa useasti epähienoihin suunnitteluratkaisuihin. Esimerkiksi jotkut järjestelmät eivät tue tarkkaa kapselointia ja mahdollistavat siten pääsyn olion tilaan muilla keinoin.

- Pysyvyysominaisuuksilla varustettu olio-ohjelmointikieli (persistent OOPL). Useasti nämä ratkaisut eivät ole täydellisiä tietokannanhallintajärjestelmiä, vaan keskittyvät tiedon pysyvyyteen eivätkä muihin tietokannoille tyypillisiin ominaisuuksiin.

- Olioilla toteutetut kerrokset. Nämä järjestelmät tuottavat oliorajapinnan olemassa olevaan tietokantajärjestelmään. Tämä vaatii oliotietomallin sovittamisen (mapping) käytössä olevan

tietokannan tietomalliin ja päinvastoin; tämä lähestymistapa voi kärsiä huonosta suorituskyvystä. Tämä muodostuu ongelmaksi myös silloin, kun oliosuuntautunutta ohjelmistokerrosta, kuten CORBAa, käytetään tietokannan liittämiseen järjestelmään.

- Tyylipuhdas oliotietokanta (full OODBMS). Nämä tietokannat on kehitetty pohjautuen pelkästään oliopohjaisiin konsepteihin ja ovat usein suorituskyvyltään parempia kuin oliokerrosarkkitehtuuriin perustuvat tietokannat. Tämä johtuu osin tavasta, jolla olion dataa käsitellään. Toisiinsa liittyvistä oliosta voidaan muodostaa rypäitä (clusters) saantiaikojen parantamiseksi. Monimutkaisten muunnosten tekemiseen eri tietomallien välillä ei tyylipuhtaassa oliotietokannassa ole tarvetta.

Machuran (1996) mukaan oliotietokannat eivät ole saavuttaneet laajempaa käyttöä perinteisissä yrityskäyttöön tarkoitetuissa tietokantasovelluksissa. Yleisesti mainittuja oliotietokantojen puutteita ovat standardien puuttuminen, riittämätön tuki arvoon perustuville kyselyille, eri näkymien (views) ja käyttökontrollin puute sekä tehokkaiden kehitystyökalujen puuttuminen. Haittapuolena on myös se, että oliotietokantoja ei voi enää pitää sovelluksista riippumattomina tietovarastoina. (Machura 1996, s. 559.) Verrattuna relaatiotietokantajärjestelmiin oliosuuntautuneiden tietokantajärjestelmien tutkimus- ja kehityshistoria on lyhyt (Huemer et al. 1995, s. 1067).

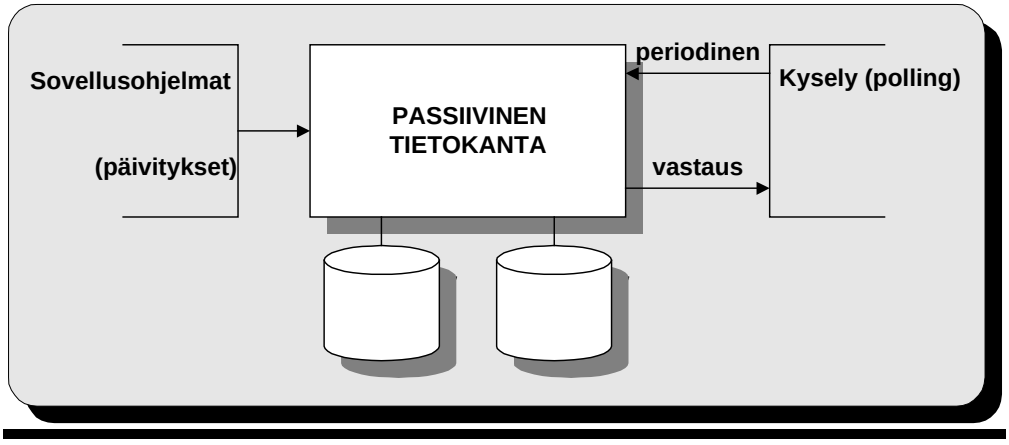
3. Joustavuutta tukevat konseptit

Tässä luvussa käsitellään joustavuutta tukevia konsepteja, joita sovellettiin pilottiprojektissa. Tarkoituksena on antaa lukijalle perustieto aihealueista. Aluksi, mitä tarkoitetaan aktiivisella tietokannalla. Aktiivisuuspiirteet toteuttava ECA-malli käsitellään tarkasti, samoin se, kuinka sitä on sovellettu muiden tutkijoiden toimesta. Sen jälkeen käsitellään aktiivisille tietokannoille esitettyjä vaatimuksia. CORBA-arkkitehtuurin perusperiaatteet käydään läpi yleisellä tasolla. Luvussa perehdytään tarkemmin arkkitehtuurin tarjoamiin transaktio- ja pysyvyyspalveluihin. Lopuksi käsitellään tietokantasovittimen arkkitehtuuria sekä OMG:n ja ODMG:n arkkitehtuurien yhdistämistä.

3.1 Aktiivisuus tietokannoissa

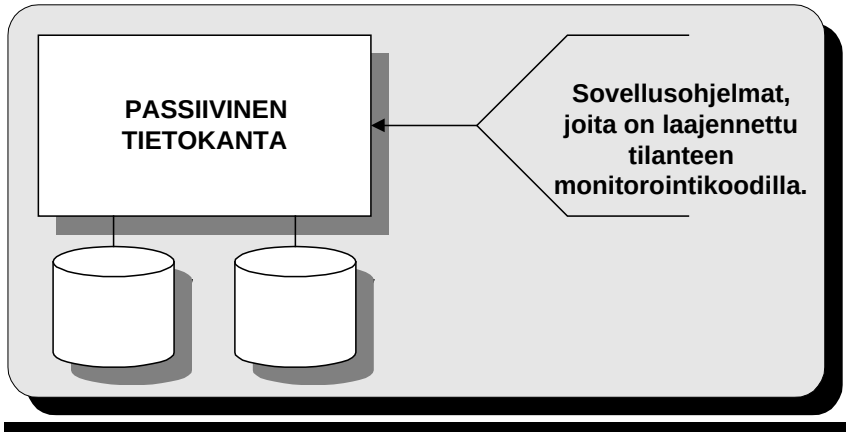
Tietokannat ovat yleensä passiivisia. Ne suorittavat kyselyjä ja transaktioita ainoastaan eksplisiittisesti, joko käyttäjän tai sovellusohjelman pyynnöstä. Moniin sovellusalueisiin, kuten prosessin ohjaukseen, voimalaitoksiin ja sähkövoiman siirtoverkkojen yms. ohjaukseen, joissa tarvitaan tarkkaa (timely) vastetta kriittisiin tilanteisiin, eivät passiiviset tietokannat sovellu kovin hyvin. Tällaisten aikakriittisten sovelluksien toimintaan täytyy monitoroida järjestelmän tiloihin määriteltyjä ehtoja, ja kun nämä ehdot totetuvat, tulee suorittaa määritellyt toiminnot, jotka mahdollisesti ovat vielä aikarajoitteisia. (Elmasri & Navathe 1994, s. 778 - 779.)

Aktiivisuuspiirteiden toteuttamiseen on ollut kaksi yleistä lähestymistapaa. Ensimmäisessä ratkaisumallissa periodisesti tietokantaan kyselyjä suorittava ohjelma tarkistaa, onko monitoroitava tilanne tapahtunut samalla, kun tietokantaa normaalisti päivittävät sovellusohjelmat suorittavat tilojen muutoksia. Tämä on vaikea toteuttaa, koska optimaalista aikaväliä tehtäville kyselyille on vaikea määritellä. Liian hidas kysely voi johtaa kriittisen tapahtuman käsittelyyn tarkoitetun aika-jakson ohittamiseen. Liian nopea kyselyjen suorittaminen johtaa puolestaan järjestelmän suorituskyvyn romahtamiseen. Tämä malli on esitettynä kuvassa 6. (Elmasri & Navathe 1994, s. 779.)



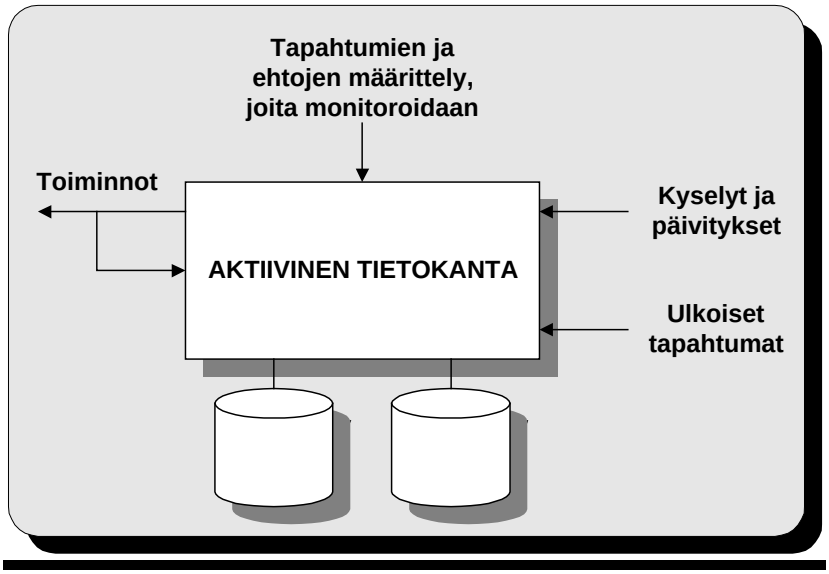
Kuva 6. Passiivinen tietokanta, joka käyttää kyselytekniikkaa aktiivisen toiminnan jäljittelemiseen (Elmasri & Navathe 1994, s. 780).

Toinen tapa toteuttaa aktiivisuus on monitoroitavan tilan tarkistamisen lisääminen erikseen jokaiseen tietokantaan päivityksiä tekevään ohjelmaan. Haittana on se, että sovellusohjelman hallittavuus ja ylläpito vaikeutuvat. Modulaarisuusperiaatteissa joudutaan kompromisseihin koodin uudelleenkäytön vaikeutuessa. Tämä malli on esitettynä kuvassa 7. (Elmasri & Navathe 1994, s. 779.)



Kuva 7. Aktiivisuuspiirteet on saatu aikaan hajauttamalla tilojen tarkistus sovelluksiin (Elmasri & Navathe 1994, s. 780).

Aktiivinen tietokanta tukee ehdon monitorointia abstraktiotasolla, jolla on kolme ominaista piirrettä. Ehdon monitoroinnin semantiikka on hyvin määritelty ja se täyttää uudentyyppisen tieto kantasovelluksen mallintamis- ja tehokkuusvaatimukset. Lisäksi monitoroinnin semantiikka yhdistyy saumattomasti käytettyyn tietokantaan. Toiminnallisesti aktiivinen tietokanta monitoroi ehtoja (conditions), jotka liipaistaan ennalta määritellyillä tapahtumilla (events). Tapahtumat kuvaavat joko tietokannan tapahtumia, kuten tiedon päivityksiä, tai muita järjestelmään liittyviä tapahtumia, kuten laitteistovikoja. Jos ehto-osan evaluoinnin tuloksena saadaan tosi, suoritetaan toiminnot (actions). Aktiivinen tietokanta mahdollistaa modulaarisuuden säilymisen ja vastauksen tapahtumaan tietyssä ajassa. Sovelluksiin ei tarvitse tehdä muutoksia. Aktiivisen tietokannan toiminta on esitetty kuvassa 8. (Elmasri & Navathe 1994, s. 779 - 780.)



Kuva 8. Aktiivinen tietokanta (Elmasri & Navathe 1994, s. 781).

3.2 ECA-konsepti

ECA-säännöt (event-condition-action) tuottavat aktiivisen tietokannan toiminallisuuden ohittaen perinteisten passiivisten tietokantojen ominaisuudet. ECA-sääntöjen suoritussemantiikka on suoraviivainen: tapahtuman esiintyessä (signaloituessa) ehto evaluoidaan ja mikäli ehto täyttyy, toiminta suoritetaan. (McCarthy & Dayal 1989, s. 216.)

Ehto (condition) on yksi sääntöön kuuluva luokka, jonka rakennetta kuvaa kytkentätapa (coupling mode) ja sarja suoritettavia kyselyitä, jotka on määritelty jollain kyselykielellä. Kytkentätapa kertoo, milloin ehto tulisi evaluoida suhteessa liipaisevaan transaktioon, jossa tapahtuma esiintyy. (Dayal et al. 1988.) Hsu et al. (1988, ks. Dayal et al. 1988) esittämässä kytkentämallissa tapahtuman ja ehdon väliseen kytkentään on neljä vaihtoehtoa:

- ehdon evaluointi välittömästi (immediate), kun liipaiseva tapahtuma on signaloitu. Tässä tapauksessa liipaiseva transaktio keskeytyy, kunnes ehto on evaluoitu ja mahdollisesti toiminto on suoritettu.
- ehdon evaluointi viivästetysti (deferred), jolloin ehto evaluoidaan liipaisevan transaktion lopussa kuitenkin ennen liipaisevan transaktion päättymistä
- erillisesti (detached), mutta kausaalisesti riippuen. Ehto evaluoidaan liipaisevasta transaktiosta erillisessä transaktiossa, mutta vasta sen jälkeen, kun liipaiseva transaktio on päättynyt.
- erillisesti, mutta kausaalisesti riippumatta. Ehto evaluoidaan erillisessä transaktiossa riippumatta liipaisevan transaktion suorituksesta.

Ehto täytetään, mikäli kaikki sen kyselyt palauttavat vastauksenaan jotakin muuta kuin tyhjän vastauksen. Kyselyiden tulokset välitetään toiminto-osalle tapahtumaan liittyvien argumenttien ohella. (Dayal et al. 1988.) Kytkentämallit määrittelevät ainoastaan liipaisevan (triggering)

transaktion suhteen liipaistuun (triggered) transaktioon. Mikäli useampia liipaistuja transaktioita on käynnissä, tulee niiden suoritusjärjestys ratkaista (Dittrich et al. 1995). Yksi mahdollisuus suoritusjärjestyksen määräämiseen ovat prioriteetit (Agraval et al. 1991, ks. Dittrich et al. 1995).

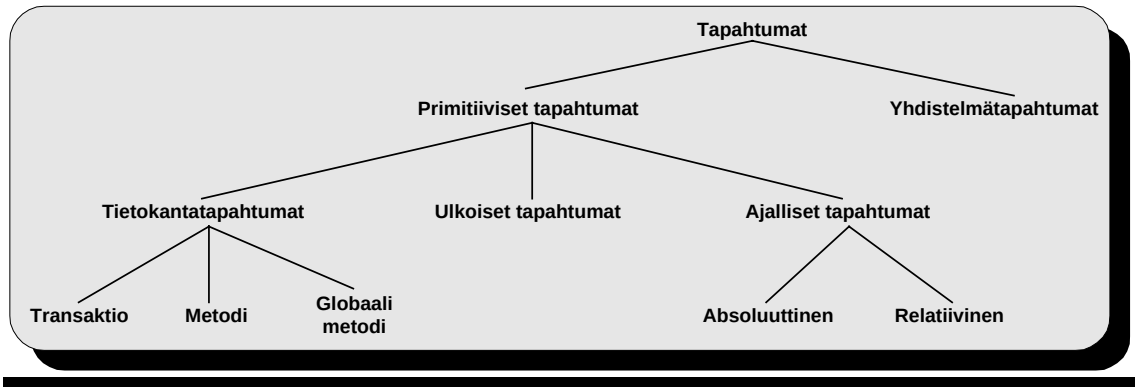
Toiminta (action) on myös sääntöön kuuluva luokka ja sillä on kytkentämalli, kuten ehdolla. Kytkentämalli kuvaa nyt ehdon evaluointiin käytetyn transaktion suhdetta suoritettaviin toimintoihin. Lisäksi toiminnolla on suoritettava operaatio. (Dayal et al. 1988.) Kytkentämallin vaihtoehdot ovat: toiminto suoritetaan välittömästi ehdon evaluoinnin jälkeen, viivästetysti ehdon evaluointiin käytetyn transaktion lopussa tai erillisessä transaktiossa, joko kausaalisesti riippuen tai riippumatta, kuten ehto-luokan tapauksessa. (Hsu et al. 1988, ks. Dayal et al. 1988). Toiminto-osaan kuuluu myös suoritettava operaatio, joka voi olla tiedon käsittelyyn käytetyn kielen ohjelma tai ulkoiselle ohjelmalle tai prosessille lähetettävä sanoma.

Sääntöön kuuluvat ehdon, toiminnon ja tapahtuman lisäksi myös aikarajoitteet ja varasuunnitelmat, mikäli aikarajoitteita ei voida täyttää, sekä säännön identifioiva arvo. Säännöllä voi olla lisänä myös muita määreitä (Dayal et al. 1988).

Tapahtuma-liipaisin mekanismi on alun perin suunniteltu suunnittelutietoa sisältävien tietokantojen pitämiseksi konsistenssissa tilassa. Tällaisissä järjestelmissä transaktiot ovat tyypillisesti pitkiä. Mekanismi yhdisti tapahtumat ja liipaisimet toimintoihin, ja esimerkiksi tietokannan yhdenmukaisuus voitiin tarkistaa. (Dittrich et al. 1986).

Gehani et al. (1992) määrittelevät tapahtuman seuraavasti: “An ‘event’ is a happening of interest.” Tapahtumat sattuvat yhtä-äkkisesti tiettynä ajanhetkenä. Oliosuuntautuneissa tietokannoissa tapahtumat ovat sidoksissa olioiden toimintoihin ja tiloihin. (Gehani et al. 1992.)

Chakravarthy et al. (1994) luokittelevat tapahtumat kehittämässään Snoop-tapahtuman määrittelykielessä kuvan 9 mukaan. Primitiiviset (primitive) tapahtumat ovat tapahtumia, jotka ovat ennalta määritelty järjestelmään. Yhdistelmätapahtumat (composite) muodostuvat yhdistelemällä primitiivisiä tapahtumia ja jo muodostettuja yhdistelmätapahtumia tapahtumaoperaattorein. Primitiiviset tapahtumat luokitellaan edelleen tietokantatapahtumiin sekä ulkoisiin (explicit/external) ja ajallisiin (temporal) tapahtumiin. Tietokannan tapahtumat liittyvät tietokannassa tapahtuviin toimintoihin, kuten transaktioihin ja metodien suoritukseen. Ajalliset (temporal) tapahtumat jakaantuvat kahteen alityyppiin, absoluuttisiin ja relatiivisiin tapahtumiin. Absoluuttinen tapahtuma on sidottu tiettyyn ajanhetkeen, ja relatiivinen tapahtuma viittaa johonkin toiseen tapahtumaan tietyn aikajakson sisällä. Ulkoiset tapahtumat havaitaan ja signaloidaan parametreineen sovellusohjelmissa tietokannan ulkopuolella; tapahtumat otetaan vastaan ja prosessoidaan järjestelmässä. (Chakravarthy et al. 1994.)



Kuva 9. Tapahtumien luokittelu (Chakravarthy et al. 1994).

Snoop-tapahtuman määrittelykielen operaattorit ydistelmätapahtumien muodostamiseen ovat taulukossa 2.

Taulukko 2. Snoop-tapahtuman määrittelykielen operaattorit yhdistelmätapahtumien muodostamiseen (Chakravarthy 1997).

Toiminta	**Operaattori**	**Kuvaus**
AND	$\wedge$	E1 $\wedge$ E2, kun E1 ja E2. Yhdistelmätapahtuma esiintyy, kun alitapahtumat E1 ja E2 ovat esiintyneet. Alitapahtumien järjestys ei vaikuta asiaan.
OR	$\vee$	E1 $\vee$ E2, kun E1 tai E2. Yhdistelmätapahtuma esintyy, kun alitapahtumista esiintyy E1 tai E2.
SEQ (sequence)	>>	E1 >> E2, kun E2 tapahtuu ja sitä ennen on tapahtunut E1. Yhdistelmätapahtuma esiintyy vain, kun alitapahtumat ovat esiintyneet määrätyssä järjestyksessä.
NOT	**!**	!(E1, E2, E3), kun E2 ei tapahdu E1:en ja E3:n välissä. Yhdistelmätapahtuma esiintyy, kun E3 tapahtuu ja sitä on edeltänyt alitapahtuma E1, mutta ei alitapahtumaa E2.
Jaksoton (aperiodic)	**A, A***	A(E1, E2, E3), aina kun E2 ja E2 on tapahtumien E1 ja E3 välissä. Yhdistelmätapahtuma esiintyy aina, kun ali tapahtuma E2 esiintyy ja sitä on edeltänyt alitapahtuma E1, mutta ei alitapahtumaa E3. A toiminnassa yhdistelmä tapahtuma voi esiintyä useita kertoja. A*-toiminnassa yhdistelmätapahtuma voi esiintyä vain kerran.
Jaksollinen (periodic)	**P, P***	P(E1, E2, E3), aina kun relatiivinen temporaalitapahtuma E2 ja kun E2 on alitapahtumien E1 ja E3 välissä. Alitapahtumalle E2 voidaan määritellä toistava aikajakso. P*-toiminnassa yhdistelmätapahtuma voi esiintyä vain kerran.
PLUS	+	E1 + [TI]. Yhdistelmätapahtuma esiintyy, kun ali tapahtuma E1 on tapahtunut ja sen jälkeen aikajakso TI on kulunut.

Chakravarthy et al. (1994) mukaan eri sovellusalueilla voi esiintyä tarvetta käsitellä yhdistelmätapahtumiin liittyviä primitiivisiä tapahtumia tavalla, joka sopii sovellusalueluokan vaatimaan semantiikkaan.

Nämä erilaiset tapahtumankäsittelykontekstit määritellään käyttäen apuna yhdistelmätapahtuman tunnistamisen aloittavia (initiator) ja päättäviä (terminator) tapahtumia. Tapoja, joilla yhdistelmätapahtumia voidaan käsitellä, on neljä: viimeisin (recent), selostava (chronicle), jatkuva (continuous) ja kertyvä (cumulative). (Chakravarthy 1997.) Taulukossa 3 ovat selostettuna eri kontekstien toiminnallisuuksien erot.

Tapahtumahistoria on tyypillinen kaikille toimiville aktiivisille tietokantajärjestelmille, ja se koostuu kaikkien määriteltyjen tapahtumien esiintymistä mahdollisesti useammista sessioista. Tämä mahdollistaa yhdistelmätapahtumien havaitsemisen, vaikka osatapahtumat olisivat esiintyneet järjestelmän eri käyttöaikoina. Tapahtumien sidonta (binding) tapahtumiin liittyviin tietoalkioihin määrittelee tapahtumien tarkkuuden ja samalla tietoalkiot, joihin ehto- ja toiminto-osat voivat viitata.(Dittrich et al. 1995.) Oliokohtainen sidonta (instance oriented) tarkoittaa sitä, että vain tiettyyn olioon, johon tapahtuma kohdistui, voidaan viitata. Sarjakohtainen sidonta (set oriented) liittyy joukkoon olioita, joihin ehto- ja toiminto-osat voivat viitata. Sidonta voi olla myös tyyppiä ennen (prior). Tällöin ehdot voivat viitata myös tilaan ennen ehdon tapahtumista. (Paton et al. 1994, ks. Dittrich et al. 1995.)

Taulukko 3. Yhdistelmätapahtumien käsittelykonstekstit (Chakravarthy et al. 1994).

Käsittelytapa	**Kuvaus**
Viimeisin (recent)	Jokaisesta alitapahtumasta huomioidaan viimeisin esiintymä, alkaen yhdistelmäsäännön tunnistamisen aloittavasta tapahtumasta. Tämän jälkeen tapahtumat poistetaan. Kaikkia primitiivisiä tapahtumia ei välttämättä käytetä.
Selostava (chronicle)	Alitapahtumista huomioidaan aina vanhin esiintymä, jonka jälkeen ne poistetaan. Samaa tapahtumaa käytetään korkeintaan kerran.
Jatkuva (continuous)	Jokainen alitapahtuma, joka osoittaa jakson alkua yhdistelmätapahtumassa, voi aloittaa alitapahtumien käsittelyn. Jokaista alitapahtumaa käytetään vähin tään kerran.
Kertyvä (cumulative)	Yhdistelmätapahtuman alitapahtumat koostuvat kaikista siihen asti tapahtuneista alitapahtumien esiintymistä, joista jokainen poistetaan yhdistelmä tapahtuman jälkeen. Alitapahtuman esiintymä osallis tuu vain yhteen yhdistelmätapahtumaan.

Chakravarthyn (1995) mukaan aktiivista toiminnallisuutta tukevien sääntöjen osista näyttää tietokantayhteisössä vallitsevan konsensus. Säännön osat on koostettu tehdyissä järjestelmissä kolmella eri tavalla. EC-A-säännössä määritellään ainoastaan toiminto- ja ehto-osat, tapahtumat määrittyvät implisiittisesti. E-CA -sääntö koostuu pelkästään tapahtuma- ja toiminto-osista, säännöt ovat toteutettuna toiminnon osana. E-C-A-säännössä kaikki kolme komponenttia on eksplisiittisesti määritelty. Yleisesti käytetyt kielet tiedon käsittelyyn eivät tue ECA-sääntöjä, joten aktiiviset tietokannat vaativat laajennuksia käytettyyn tietomalliin. (Chakravarthy 1995.)

3.2.1 ECA-konseptin käyttäminen

HiPAC-projekti (Dayal et al. 1988) toimi aktiivisten tietokantojen pioneerina 1980-luvulla. Tätä ennen aktiivisia tietokantoja ei laajalti tunnettu, mutta useita ehdotuksia aktiivisuudesta ja aktiivisten ominaisuuksien toteutuksista tietokantoihin oli olemassa jo 1970-luvulla ja 1980-luvun alussa. (Widom & Ceri 1996, s. 6.) Ensimmäinen HiPAC-arkkitehtuurilla toteutettu sovellusohjelma oli arvopaperikaupan apuna käytettävä ohjelmisto SAA (securities analyst's assistant), jonka tarkoituksena oli informaation välittäminen analyytikon näytölle ja kauppojen automaattinen suorittaminen annettujen ohjeiden mukaan. Järjestelmä koostui seuraavista kolmesta ohjelmasta: ticker, display ja trader. Ticker suoritti hintatietojen päivitykset ulkoisesta lähteestä, display näytti pelkästään tietoa analyytikon työasemalla ja trader-osa hoiti varsinaisen kaupankäynnin. Jokaisesta ohjelmasta saattoi olla useampia kopioita käynnissä yhtäaikaisesti. Ohjelmat liittyivät toisiinsa aktiivisen tietokannan säännöillä, jotka jakautuivat kahteen ryhmään, näyttöihin- ja kaupankäytiin liittyviin sääntöhin, toiminto-osissa suoritettavien operaatioiden mukaisesti. Tiedon välitys eri näyttöihin tapahtui jokaiselle näytölle omalla päivitystapahtumakohtaisella säännöllä. (McCarthy & Dayal 1989.)

Järjestelmän toteutuksessa huomattiin, että sovellusohjelmien välille ei muodostunut suoraa vuorovaikutusta, vaan vuorovaikutus tapahtui sääntöjen kautta. Ohjauslogiikka toteutui sääntöihin, ja sovellukset muodostuivat yksinkertaisiksi palvelimiksi. Suurin osa säännöistä sisälsi toiminto-osissa kutsuja sovellusohjelmiin, ei niinkään tietokantaan kohdistettuja toimintoja. Sovelluksen käyttäytymisen muuttamiseksi muutetaan pelkästään sääntöjä ohjelmiston sijaan. Sovelluksen korkean tason logiikka voidaan koodata sääntöihin ohjelmiston sijaan, mikä tekee sovelluksesta modulaarisemman ja helpommin muutettavan. Ohjelmiston yksinkertaistuessa tuotetaan paljon sääntöjä. (McCarthy & Dayal 1989.)

Kappelin (et al. 1994) kehittämä TriGS-järjestelmä on toiminut tämän tutkielman ECA-vuorottajan esikuvana. TriGS-järjestelmä (Trigger System for GemStone) pohjautui GemStone-oliotietokantaan ja OPAL-ohjelmointikieleen (Kappel et al. 1994). Aktiivisia tietokantoja on toteutettu myös perinteisillä relaatiotietokannoilla. Esimerkkejä tällaisista järjestelmistä ovat *Ariel* (Hanson 1996) ja *Starburst* (Widom 1996).

3.3 Aktiiviselle tietokannalle asetettuja vaatimuksia

Seuraavassa luetellaan Dittrichin et al. (1995) esittämiä vaadittavia ominaisuuksia, jotka tulee täyttää, jotta tietokantaa voitaisiin kutsua aktiiviseksi tietokannaksi. He jakavat piirteet olennaisiin ominaisuuksiin, joita täytyy tukea ja optionaalisiin (toivottaviin) ominaisuuksiin. Aktiivisen tietokannan olennaisia piirteitä ovat:

1) Aktiivinen tietokanta on tavallinen tietokanta. Kaikki toiminnallisuus, joka vaaditaan passiiviselta tietokannalta, vaaditaan myös aktiiviselta tietokannalta. Mikäli aktiivisuuspiirteitä ei käytetä, tulee aktiivisen tietokannan toimia kuten passiivinen tietokanta.

2) Aktiivinen tietokanta tukee ECA-sääntöjen hallintaa ja määrittelyä. Passiivista tietokantaa laajennetaan siten, että se tukee reaktiivista käyttäytymistä, jonka tulee olla käyttäjän määriteltävissä. Liityntää tietokantaan, esimerkiksi datan määrittelykieltä (data definition language) täydennetään tai laajennetaan operaatioilla sääntöjen määrittelyyn.

2a) Aktiivisella tietokannalla tulee olla keinot tapahtumien, ehtojen ja toimintojen määrittelyyn. Tapahtumat tulee olla määriteltävissä tapahtuma-ehto-pareina. Lisäksi vaaditaan, että aktiivinen tietokanta tukee tapahtumien eksplisiittistä määrittelyä käyttäen tietokannan liityntää. Mikäli eksplisiittisesti määriteltäviä tapahtumia ei tueta, ECA-säännöistä muodostuu pelkästään tehtävien sisäinen toteutusmekanismi, joka voitaisiin toteuttaa myös passiivisesti, jos yleinen tuki reaktiivisuudelle puuttuisi. Aktiivisen tietokannan tulee siis tukea tapahtuman määrittelemistä, jotta pystytään määrittelemään, milloin reaktiivisuus tulee suorittaa. Tämä erottaa aktiivisen tietokannan muista sääntöpohjaisista tietokannoista.

Jossain tapauksissa voi olla hyötyä tapahtumien implisiittisestä määrittelystä tietokannan toimesta. Mikäli on tarkoituksenmukaista, tulee olla mahdollista määritellä ennen- ja jälkeen-tapahtumat; esimerkiksi tietokantatoiminnoissa ennen-tapahtuma signaloitaisiin juuri ennen toiminnon suorittamista. Mikäli säännölle ei ole määritelty tapahtumaa, ehto-osa voidaan jättää pois. Tässäkin tapauksessa tulisi olla mahdollista määritellä ehdot osana toimintoja. Kysymyksessä on tapahtuma-toiminto-sääntö.

Kaikki aktiivisen tietokannan osat tulisi olla integroitavissa passiiviseen tietomalliin. Tuettujen tapahtumatyyppien tulisi ainakin tukea datan määrittelykielen (DML) operaatioita ja transaktiokäskyjä. Esimerkiksi jonkin relaation, josta ollaan kiinnostuneita, päivitys voidaan määritellä tapahtumaksi. Ehdot tulisi olla määriteltävissä tietokantaan kohdistuvina kyselyinä, joiden yhteydessä tulisi olla käytössä myös tietokannan tarjoamat palautusominaisuudet (retrieval facilities). Toiminto-osat ovat periaatteessa mitä tahansa suoritettavaa koodia, joissa tulisi olla mahdollista käyttää vähintään datan käsittelykielen (data manipulation language) ja transaktion ohjaamisen käskyjä; esimerkiksi liipaisevan transaktion keskeyttäminen. Lopuksi tulee erottaa erityyppiset tapahtumat niiden esiintymisestä.

2b) Aktiivisen tietokannan tulee tukea sääntökannan muuttamista ja sääntöjen hallintaa. Sääntökannan muodostaa tiettynä ajanhetkenä määritelty sääntöjoukko. Tietokannan tulee hallita sääntökantaa riippumatta siitä, mihin säännöt on talletettu. ECA-sääntöjen määrittely on osa tietokannan sisältämää metatietoa ja itse tietokantaa. Aktiivisen tietokannan tulee mahdollistaa uusien ECA-sääntöjen määrittely ja vanhojen sääntöjen poistaminen sekä olemassa olevien sääntöjen eri osien muuttaminen. Lisäksi säännöt on voitava kytkeä päälle ja pois. Tieto olemassa olevista säännöistä määrittelyineen tulee säilyttää. Tämän tulee näkyä myös käyttäjille ja sovelluksille.

3) Aktiivisella tietokannalla on suoritusmalli (execution model). Tapahtumien esiintymät tulee havaita ja niihin liittyvät ehdot pitää pystyä evaluoimaan. Aktiivisen tietokannan tulee pystyä

toimintojen suorittamiseen, ja sääntöjen suoritussemantiikka tulee olla hyvin määritelty. Lisäksi sääntöjen suoritusjärjestys tulee olla määriteltävissä tai ennalta määrätty.

3a) Aktiivisen tietokannan tulee havaita tapahtumien (events) esiintymät. Ideaalitapauksessa aktiivinen tietokanta havaitsee kaikenlaiset tapahtumat automaattisesti ilman käyttäjän tai sovelluksen tekemää signalointia. Muussa tapauksessa, mikäli sovellusohjelmoijat tai käyttäjät ovat vastuussa kaikentyyppisten tapahtumien signaloinnista, järjestelmä on passiivisen tietokannan syntaktinen muunnos.

3b) Aktiivisen tietokannan tulee pystyä ehdon evaluointiin. Tapahtuman havaitsemisen jälkeen tulee ehdot pystyä evaluoimaan, lisäksi tulee pystyä välittämään informaatio tapahtumista ehto-osille, esimerkiksi tieto juuri tiettyyn olioon tai tiettyihin relaation riveihin kohdistuneesta tapahtumasta. Ehdossa tulisi myös pystyä suorittamaan kyselyjä tietokannan tilasta.

3c) Aktiivisen tietokannan tulee pystyä suorittamaan toimintoja (actions). Tapahtuman jälkeen ehto-osan toteutuessa tietokannan tulee pystyä suorittamaan toimintoja. Tietoa tulee pystyä välittämään ehto-osalta toiminto-osalle. Toiminnot tulisi myös pystyä suorittamaan liipaisevan (triggaavan) transaktion osana rinnakkaisuudenhallinnan alaisuudessa ja sen palautettavissa.

3d) Aktiivisella tietokannalla on hyvin määritelty suoritussemantiikka. Suoritussemantiikka on hyvin määritelty, kun tapahtumien käsittely, tunnistaminen ja signalointi ovat hyvin määriteltyjä. Mikäli aktiivinen tietokanta tukee yhdistelmätapahtumia (composite events), tapahtumien käsittely (event consumption) määrittelee, mistä tapahtumakomponenteista yhdistelmätapahtuma koostuu ja kuinka sen tapahtumaparametreja käsitellään. Säännön suorittamisella tulee olla selkeä semantiikka, miten, milloin ja mitä tietokannan tilaan liittyviä ehtoja evaluoidaan ja mitä toiminto-osia suoritetaan tietokannan asettamien rajoitusten mukaisesti. Ehdon evaluoinnin ja toiminto-osan suorituksen suhde liipaisevaan transaktioon (coupling mode) tulee määritellä. Vähintään tulee toteuttaa välitön- (immediate) ja/tai siirretty (deferred) kytkentämalli. Tapahtumat voivat olla joko instanssikohtaisia tai liittyä olioiden joukkoon, mikä tulee määritellä.

Lopuksi täytyy määritellä, mikä tietokannan tila on näkyvissä ehdon evaluoinnille ja toiminnon suorittamiselle. Tila voi olla joko säännön suorituksen aikainen (actual) eli tehtyjen tilanmuutosten jälkeinen tila tai tila ennen muutoksia (prior). Useamman tilan näkeminen voi olla tarpeellista ehto- ja toiminto-osissa. Tällöin pystytään tarkastelemaan tilaa ennen ja jälkeen muutosten. On suotavaa muttei pakollista tarjota useampia vaihtoehtoja jokaiseen tai osaan esitetyistä ominaisuuksista. Näin lisätään käyttäjän mahdollisuuksia määritellä tekemiensä sääntöjen suoritussemantiikka.

3e) Sääntöjen suoritusjärjestyksen (conflict resolution) tulee olla joko käyttäjän määriteltävissä tai ennalta määrätty. Mikäli samaan tapahtumaan on sidottuna useita, tulee tietokannan pystyä määrittelemään järjestys, jolla säännöt suoritetaan. Säännön tekijällä tulee olla mahdollisuus määrätä suoritusjärjestys, esimerkiksi käyttäen prioriteetteja. Mikäli näin ei haluta menetellä, tulee aktiivisen tietokannan määritellä suoritusjärjestys tai suorittaa säännöt epädeterminisesti.

Dittrich et al. (1995) määrittelevät lisäksi aktiivisille tietokannoille seuraavia valinnaisia ominaisuuksia: aktiivisen tietokannan tulisi esittää informaatio ECA-säännöistä tietomallinsa mukaisesti pakottamatta opettelemaan uutta esitystapaa ja sen tulisi tukea jotain ohjelmointiympäristöä. Jotta aktiivisen tietokannan käyttö olisi tehokasta, tulisi seuraavat työkalut toimittaa tietokannan mukana: sääntöselain-, säännön suunnittelu-, sääntökannan analysointi-, virheenetsintä-, ylläpito-, jäljitys- ja suorituskyvyn optimointityökalut. Lisäksi aktiivisen tietokannan tulisi olla säädettävissä (tunable). Aktiivinen tietokantaratkaisu ei saa olla merkittävästi huonompi suorituskyvyltään verrattuna passiiviseen järjestelmään. Kokemus aktiivisten tietokantojen suorituskyvyn systemaattisesta mittaamisesta on vähäinen. (Dittrich et al. 1995.)

Dittrichin et al. (1995) mukaan aktiiviselta tietokannalta vaadittuja ominaisuuksia ei ole kuitenkaan mahdollista määritellä yleisesti, koska niitä käytetään erilaisiin tarkoituksiin vaatien niiden käyttöön erilaisia mahdollisuuksia ja ominaisuuksia. He luokittelevat tämän takia aktiiviset tietokannat tarkemmin:

- aktiivisen tietokannan roolin mukaan järjestelmässä (valvonta tai ohjaus)
- järjestelmän integrointitason mukaan (hetero- tai homogeeninen).

Järjestelmää valvova aktiivinen tietokanta vertaa tietokantaoperaatiopyyntöjä itse tietokannan tilaan tai toisinpäin ja suorittaa yksinkertaisia toimintoja. Metasäännöt koko järjestelmän toiminnasta ovat käyttäjällä mutta eivät välttämättä ole ilmaistavissa ECA-säännöillä. Tällainen järjestelmä on käyttökelpoinen tavallisten tietokantatoimintojen toteuttamisessa. Järjestelmää ohjaava aktiivinen tietokanta pystyy kontrolloimaan koko sovellusympäristön toimintaa eikä pelkästään tietokannan tilaa. Se pystyy ‘triggaamaan’ ulkoisten sovellusohjelmien funktioihin.

Homogeenisyydellä tarkoitetaan tässä yhteydessä järjestelmää, jonka kaikki osat ovat käytettävän aktiivisen tietokannan sovelluksia jakaen yhteisen kaavan (schema) ja tietokannan. Muutoin järjestelmä on heterogeeninen, erityisesti silloin, kun aktiivisella tietokannalla ohjataan muille alustoille toteutettuja järjestelmiä. Muodostuu kolme yhdistelmää, koska valvonta-heterogeeninen-yhdistelmää ei voida pitää mielekkäänä. (Dittrich et al. 1995, s. 12 - 13.). Yhdistelmät ovat

1. valvonta homogeenisessä järjestelmässä (yksinkertaisin)
2. ohjaus homogeenisessä järjestelmässä
3. ohjaus heterogeenisessä järjestelmässä (laajin).

Jokaisen luokan tulee sisältää edellisen luokan ominaisuudet, ja kolmannelle luokalle asetetut vaatimukset on esitetty taulukossa 4. Kolmannen luokan aktiiviset tietokannat kykenevät yhdistämään heterogeenisiä ja autonomisia järjestelmiä. Näin ollen tulee olla mahdollista havaita tilanteet muissa järjestelmissä ja mahdollisesti myös ulkoisissa laitteissa.

Taulukko 4. Yhteenveto aktiivisen tietokannan ominaisuuksista heterogeenisen järjestelmän ohjauksessa (Dittrich et al. 1995, s. 14 - 16).

Ominaisuus	**Toteutus**
Tapahtumat (2a)	DML-toiminnot, ulkoiset tapahtumat, yhdistelmä tapahtumat
Ehdot (2a)	boolean-tyyppinen funktio, kyselyt (predicates) tietokannan tilasta, muut kyselyt
Toiminnot (2a)	DML-toiminnot, käyttäjälle ilmoittaminen, ulkoiset ohjelmat, varatoiminnot
Säännön muodostaminen (2b)	luominen/poistaminen + muuttaminen ja tapahtuma historiaan mukautuminen, päälle/pois
Tapahtumien käsittely (3d)	valinnainen, mukaan lukien kertova (chronicle)
Kytkentätapa (3d)	välitön, siirretty, erotettu + syyperäiset riippuvuudet
Suoritus (3d)	paikallisesti ohjattuna ja hajautetusti ohjattuna

Luokan tärkein ominaisuus on vahva säännön suoritusmekanismi, koska suoritettavia toimintoja ei voida välttämättä suorittaa paikallisen transaktionhallinnan alaisuudessa. Lisäksi, mikäli tällaista järjestelmää aiotaan soveltaa reaaliaikakäytössä, tulisi sääntöjen suorituksessa tukea aikarajoituksia. Mikäli aikarajoituksia ei voida täyttää, tulisi suorittaa varatoiminnot. Kolmannen tason aktiivinen tietokanta edistää välitason ohjelmiston toteuttamista tietokantaan. Nämä tietokannat on tarkoitettu käytettäviksi mahdollisesti heterogeenisissa, löysästi kytketyissä järjestelmissä. Aktiivisen toiminnallisuuden yhdistämisen tulisi olla mahdollista tällaisiin ympäristöihin tarkoitettuihin ohjelmistoarkkitehtuureihin, erityisesti OMG:n CORBA-arkkitehtuuriin. (Dittrich et al. 1995.)

3.4 CORBA-Arkkitehtuuri

OMG (Object Management Group, Inc.) on yli 500 jäsenen muodostama kansainvälinen organisaatio, johon kuuluu informaatiojärjestelmien toimittajia sekä ohjelmistojen kehittäjiä ja käyttäjiä. Järjestö on perustettu vuonna 1989 ja sen päätarkoituksia ovat heterogeenisten hajautettujen ympäristöjen oliopohjaisen ohjelmiston siirrettävyys, yhteensopivuus ja uudelleenkäyttö. (The Common--. 1995, iii.) CORBA-määrittely julkistettiin lokakuussa 1991 (Baker 1997, s. 14).

Mowbray & Zahavi (1995, s. 2) mukaan CORBA on järjestelmien integrointiin luotu teollisuusstandardi, jonka tärkeimpiä etuja ovat

- hajautetun tietojenkäsittelyn ja sovellusten integroinnin yksinkertaistuminen

- oliopohjaisuus

- sellaisen teknologian vakauttaminen, jota monet kaupalliset valmistajat tukevat.

Käsitteellisesti CORBA sijoittuu OSI-mallin (Open Systems Interconnection) ylimälle tasolle, sovellustasolle (kuva 10). CORBA eristää asiakkaan ja ohjelmoijat informaatiojärjestelmän hajautetuista heterogeenisista piirteistä. OMG:n IDL-kieli tuottaa käytetystä ohjelmointikielestä ja käyttöjärjestelmästä riippumattoman liitynnän. Korkeammasta abstraktiotasosta johtuen ohjelmoijan ei tarvitse välittää alemman tason protokollista. Ohjelmoijan ei myöskään tarvitse huomioida palvelimen käyttöjärjestelmää, laitteistoa, sijaintia tai käynnistystilaa. Myös asiakkaan ja palvelimen integrointiin liittyviä asioita on yksinkertaistettu. Esimerkiksi osapuolien ei tarvitse enää huolehtia välitettävän datan tavujärjestyksestä. Tietoturvaan, tiedon yksityisyyden ja koskemattomuuden saavuttamiseen, on tehty paljon tutkimustyötä. (Mowbray & Zahavi 1995, s. 22.)

Kuva 10. OSI-malli ja CORBA (Mowbray & Zahavi 1995, s. 22).

CORBA-arkkitehtuuri on viitemalli, joka koostuu seuraavista neljästä komponentista (The Common--. 1995, s. iv - v.):

- oliokutsujen välitin (object request broker); mahdollistaa olioiden läpinäkyvän kutsujen tekemisen ja vastaanottamisen hajautetussa ympäristössä. Kutsujen välitin toimii perustana sovellusten rakentamiselle hajautetuista olioista sekä yhteentoimivuudelle sovellusten välillä homo- ja heterogeenisissä ympäristöissä.

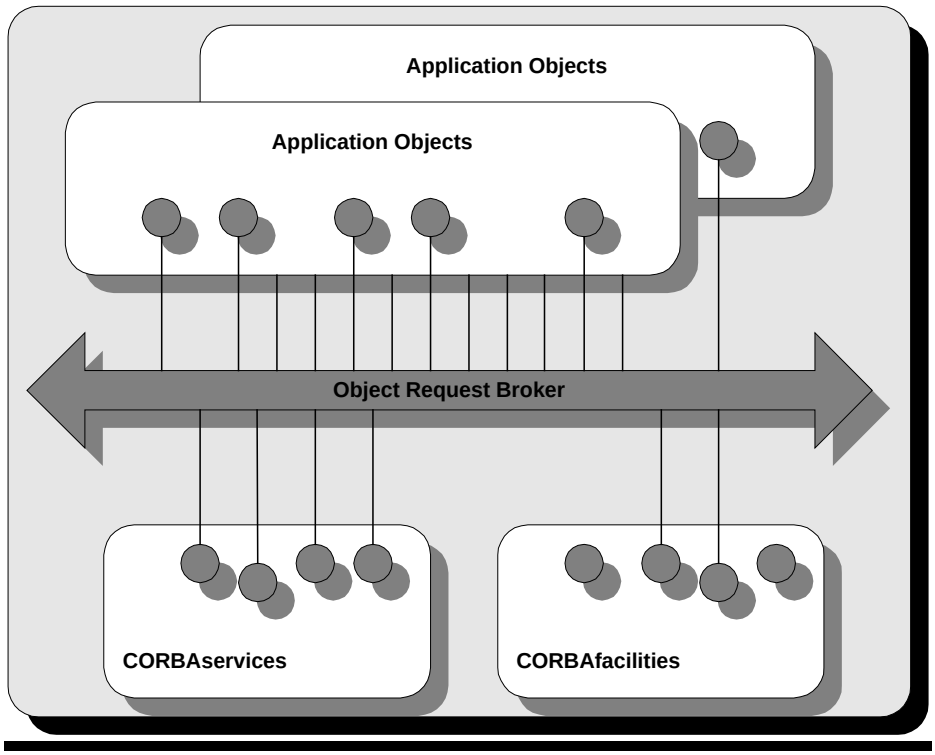
- oliopalvelut (object services); kokoelma erilaisia palveluita, liityntöjä ja olioita, jotka muodostavat perustoiminnot olioiden toteuttamiseksi ja käyttämiseksi. Palvelut ovat aina sovellusalueesta riippumattomia ja välttämättömiä hajautetun sovelluksen rakentamiseksi.

Esimerkiksi elinjaksopalvelu (life cycle service) määrittelee tavat olioiden luomiseen, poistamiseen, siirtämiseen ja kopiointiin, mutta se ei määrittele sitä, kuinka oliot tulee toteuttaa sovelluksessa.

- yleiset lisäpalvelut (common facilities); sovellusten jaettavissa oleva kokoelma palveluja, jotka eivät ole niin perustavaa laatua olevia palveluja kuin oliopalvelut. Esimerkiksi sähköpostipalvelu tai järjestelmän hallintaan liittyvät palvelut voitaisiin luokitella lisäpalveluiksi.

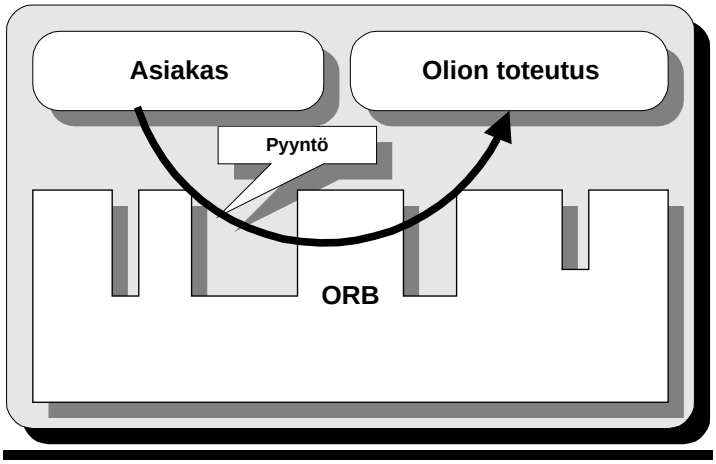
- sovellusoliot (application objects); joko oman kehitystyön kautta muodostuneita tai ulkoisen toimittajan tekemiä tuotteita, olioita, jotka kontrolloivat omia liityntöjänsä. Näitä ei ole standardoitu OMG:n toimest,a vaan ne vastaavat perinteistä järjestelmän sovellusten käsitettä.

Oliokutsujen välitin, oliopalvelut ja lisäpalvelut muodostavat olioiden hallinta-arkkitehtuurin OMA (object management architechture), joka on esitettynä kuvassa 11 (Baker 1997, s. 11).



Kuva 11. Olioiden hallinta-arkkitehtuuri (Baker 1997, 11).

Kuvassa 12 nähdään asiakkaan tekemän pyynnön välittäminen ORB:n kautta olion toteutukselle. Asiakas on entiteetti, joka haluaa suorittaa olion jonkin operaation. Olion toteutus tarkoittaa olion toteuttavaa dataa ja koodia. ORB on vastuussa kaikista mekanismeista, jotka tarvitaan olion toteutuksen löytämiseen ja valmistamiseen sekä tiedon välittämiseen pyynnön suorittamiseksi. Asiakkaan näkemä liityntä on täysin riippumaton siitä, missä olio sijaitsee tai millä ohjelmointikielellä se on toteutettu tai mistä tahansa muusta seikasta, mikä ei liity suoraan olion liityntään. (The Common--. 1995, s. 2-1 - 2-2.)



Kuva 12. Pyynnön välittäminen ORB:n kautta (The Common--. 1995, s. 2 - 1).

Asiakkaan ja palvelimen väliset metodikutsut voivat olla kahden tyyppisiä; joko staattisia tai dynaamisia. Molemmissa tapauksissa metodikutsu suoritetaan käyttäen apuna olioviittausta (object reference) palvelun suorittavaan olioon. Palvelimelle molemmat kutsutyypit näyttävät samanlaisilta. Olion staattinen liityntä muodostuu IDL-esikääntäjän luomista tynkätiedostoista (stub). Staattisen liitynnän käyttö soveltuu ohjelmiin, joiden käytettävät operaatiot tiedetään käännösaikana. (Orfali et al. 1997, s. 54.) Dynaaminen kutsuliityntä (DII) mahdollistaa asiakkaan käyttää mitä tahansa kohdeoliota ajon aikana ja dynaamisesti kutsua sen operaatioita ilman esikäännettyjä tynkätiedostoja ja käännösaikaista tietämystä. Asiakas selvittää tarvittavan liityntäinformaation kutsuhetkellä (Orfali et al. 1997, s. 61.)

Normaalisti operaatiokutsun suorittaja ei palvele sille tulevia metodikutsuja, kun sen tekemää metodikutsua suoritetaan kohdeoliossa (Baker 1997, s. 48). CORBA on synkroniseen pyyntövastauskäyttöön tarkoitettu protokolla. Metodin kutsumisen jälkeen odotetaan, kunnes sen suoritus päättyy. Mikäli ei haluta jäädä odottamaan kutsutun metodin päättymistä, tulee jokainen kutsu suorittaa omassa säikeessään. CORBA-arkkitehtuuri tarjoaa myös muita mahdollisuuksia asynkronisuuden saavuttamiseksi. Metodin voi määritellä joko yksisuuntaiseksi (oneway) tai voi käyttää DII:n viivästettyä synkronista moodia tai lopuksi voi käyttää tapahtumapalvelua. (Orfali et al. 1997, s. 257 - 258.) CORBA-standardi ei kuitenkaan vaadi yksisuuntaisilta operaatioilta sulkematonta (non-blocking) toiminnallisuutta ja niiden liikakäyttöä tulisi välttää. Mikäli tapahtumanvälitykseltä halutaan luotettavuutta ja sulkemattomuutta, tulisi sovelluksessa käyttää tapahtumapalvelua. (Baker 1997, s. 48 - 49.)

Nykyisten ORB-toteutusten puutteita ovat CORBA-palvelujen Java-toteutusten sekä asynkronisen sanomanvälityksen (MOM) puuttuminen. Muita jatkuvia kysymyksiä ORB-toimittajille ovat palvelimen skaalautuminen suuriin järjestelmiin, kuorman tasaus ja vikasietoisuus. (Mowbray & Malveau 1997, s. 249 - 250.) Sanomanvälityspalvelu (messaging service) on todennäköisesti tulossa CORBA 3.0 -määrittelyihin (Mowbray & Malveau 1997, s. 259).

3.4.1 CORBA-arkkitehtuurin tarjoamia palveluja

CORBA-palvelut määrittelevät yhteiset standardoidut palvelut kaikille CORBA-toteutuksille, jotka parantavat sovellusten välistä yhteistoimintaa. Nämä määrittelyt ovat osa olioiden hallinta-arkkitehtuuria ja ne on suunniteltu ja kehitetty toimimaan yhdessä. Ideaalitapauksessa koodi, joka toteuttaa jonkin palvelun, olisi siirrettävissä toisiin ORB-ympäristöihin. Näiden ympäristöjen tulee tukea samaa IDL:stä ohjelmointikieleen -muunnosta ja samoja CORBA-palveluja. CORBA-palvelut edistävät koodin uudelleenkäyttöä ja teknologian muuttumista, kun protokollat ja toteutuksen yksityiskohdat kehittyvät riippumatta CORBAn ja sovellusten omista liitynnöistä. (Mowbray & Malveau 1997, s. 210.)

Baker (1997, s. 11 - 12) jaottelee palvelut kolmeen eri ryhmään, hajautettuihin järjestelmiin, tietokantoihin liittyviin ja yleisiin palveluihin. Hajautettuihin järjestelmiin liittyviä palveluja ovat

- nimeämispalvelu (naming); mahdollistaa olion paikallistamisen asiakkaan toimesta käyttäen oliolle annettua nimeä. Käyttämällä tätä palvelua palvelin voi rekisteröidä minkä tahansa olioistaan antamalleen nimelle.

- tapahtumapalvelu (event); joko asiakas tai palvelin voi lähettää tapahtumasanoman usealle vastaanottajalle. Asiakkaat ja palvelimet voidaan irrottaa toisistaan, koska sanomat voidaan väliaikaisesti tallettaa tapahtumapalvelimelle.

- turvallisuuspalvelu (security); mahdollistaa yksittäisten olioiden tai olioryhmien suojauksen siten, että ainoastaan ne käyttäjät, joilla on oikeus käyttää ko. olioita, saavat käyttää niiden operaatioita. Tietoliikennne voidaan myös kryptata.

- kaupankäyntipalvelu (trading); mahdollistaa olion paikallistamisen asiakkaan antamin rajoituksin.

Tietokantoihin liittyvät palvelut ovat

- rinnakkaisuuspalvelu (concurrency); kontrolloi olion yhtäaikaista käyttöä lukitusmekanismilla

- ominaisuuspalvelu (property); mahdollistaa nimi-arvo-parien liittämisen mihin tahansa olioon

- transaktiopalvelu (transaction); kontrolloi tapahtumia kaksivaiheisella protokollalla. Tapahtumat voivat kohdistua useampiin tietokantoihin, jotka voivat tyypiltään olla joko samanlaisia tai erilaisia.

- yhteyspalvelu (relationship): mahdollistaa yhteyksien muodostamisen ja hallinnan eri olioiden välillä

- kysely (query); mahdollistaa kyselyjen suorittamisen oliokokoelmaan. Palvelussa ei määritellä uutta kyselykieltä vaan käytetään olemassa olevia kyselykieliä.

- pysyvyyspalvelu (persistent); määrittelee abstraktin kehyksen olion ja tietokannan väliseen kommunikointiin, jotta olio saataisiin talletettua ja tarvittaessa palautettua tietokannasta.

- ulkoistamispalvelu (externalization; mahdollistaa olion kopioinnin käyttämällä apuna tavuvirtaa (stream of bytes).

Yleisiin palveluihin kuuluvat elinjaksopalvelu ja lisensointipalvelu, joka mahdollistaa ohjelmiston suorittamiseen tarvittavan lisenssin tarkistamisen palvelimen toimesta. Lisäksi on määritelty vielä aikapalvelu, jota käytetään ajan hakemiseen tai kutsujen ajastamiseen. (Baker 1997, s. 11 - 12.)

CORBA-arkkitehtuurin lisäpalvelut tuottavat korkeamman tason palveluita sovelluksille CORBA-palveluiden tuottaessa palveluita olioryhmille tai yksittäiseille olioille. Lisäpalvelut jaetaan horisontaalisiin ja vertikaalisiin lisäpalveluihin. Horisontaalisia lisäpalveluja voidaan käyttää millä tahansa sovellusalueella, kun taas vertikaaliset lisäpalvelut ovat erikoistuneet yksittäisiin markkinasegmentteihin. (Baker 1997, s. 13.)

Transaktiopalvelu, jota yleisesti kutsutaan oliotransaktiopalveluksi (OTS), tukee tietokannoille tyypillistä transaktion käsitettä, ‘työn yksikköä’, hajautetussa CORBA-järjestelmässä. Asiakas voi aloittaa transaktion, muuttaa olioiden tilaa ja lopulta päättää transaktion hyväksymisestä tai hylkäämisestä. OTS transaktiot tukevat yleisiä ACID-ominaisuuksia: (Baker 1997, s. 394 - 395.)

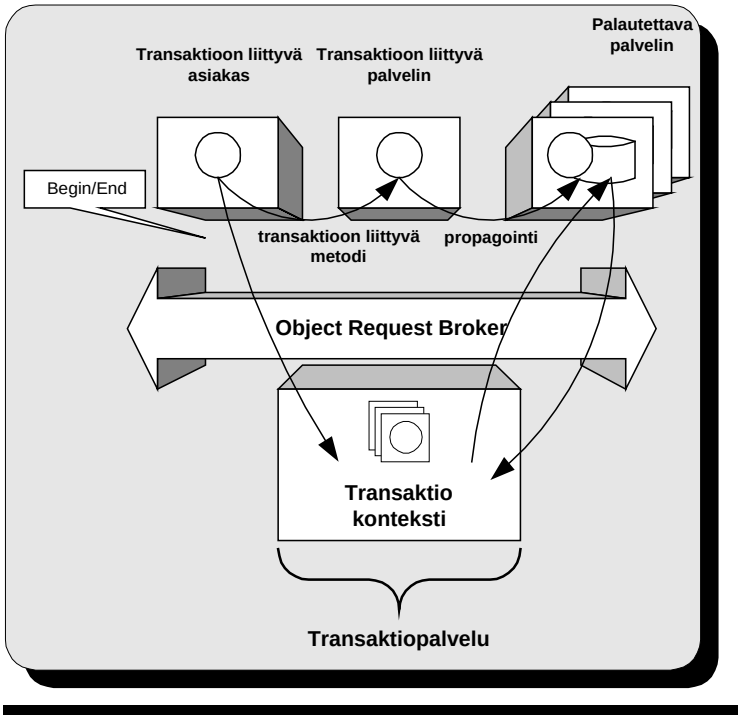
- Atominen; kaikkien tehtyjen muutosten onnistuminen, ei koskaan muutosten alijoukko.

- Konsistentti; transaktion tulisi muuttaa dataa konsistenssista tilasta toiseen konsistenttiin tilaan.

- Eristetty; yksikään asiakas, pois lukien koodi transaktion sisällä, ei saa nähdä transaktion muuttamaa dataa ennen kuin transaktio on päätetty.

- Kestävä; hyväksytyn transaktion tulokset ovat pysyviä ja niitä ei hukata, lukuun ottamatta katastrofaalista epäonnistumista.

OTS mahdollistaa transaktion aloittamisen asiakkaan toimesta, ja sen jälkeen asiakas voi kutsua järjestelmään hajautettujen olioiden operaatioita ja attribuutteja. Kaikki suoritetut kutsut assosioidaan implisiittisesti kutsuvan asiakkaan transaktioon. OTS-terminologiassa asiakkaan transaktiokonteksti propagoituu implisiittisesti niissä kutsuissa, joita asiakas tekee transaktio-ominaisuuksilla lisättyyn olioon. (Baker 1997, s. 395.) Toiminnallisuus on esitettynä kuvassa 13.



Kuva 13. Oliotransaktiopalvelun osat (Orfali et al. 1997, s. 141).

Orfalin et al. (1997) mukaan transaktioon liittyvillä olioilla voi olla jokin seuraavista kolmesta roolista: transaktio-ominaisuuksilla lisätty asiakas, transaktio-ominaisuuksilla lisätty palvelin tai palautettava palvelin.

- Transaktio-ominaisuuksilla varustettu asiakas tekee sarjan metodikutsuja, jotka merkitään begin/end-metodikutsuilla. Näiden merkkien väliin jäävät metodit voivat kutsua joko transaktio-ominaisuuksilla varustettuja oliota tai tavallisia olioita. ORB sieppaa begin-kutsun ja ohjaa sen transaktiopalvelulle, joka muodostaa oliolle transaktiokontekstin. Tämän jälkeen asiakkaan kutsumiin metodeihein lisätään tämä konteksti, joka propagoituu kaikkeen liikenteeseen transaktioon liittyvien osapuolien välillä. ORB puuttuu asiaan myös silloin, kun asiakas tekee transaktion hyväksynnän tai palauttamisen. Asiakkaalle tämä toiminnallisuus on läpinäkyvää, sen tarvitsee ainoastaan aloittaa transaktio, kutsua tarvitsemiaan metodeja ja joko hyväksyä tai peruuttaa transaktio. (Orfali et al. 1997, s. 140.)

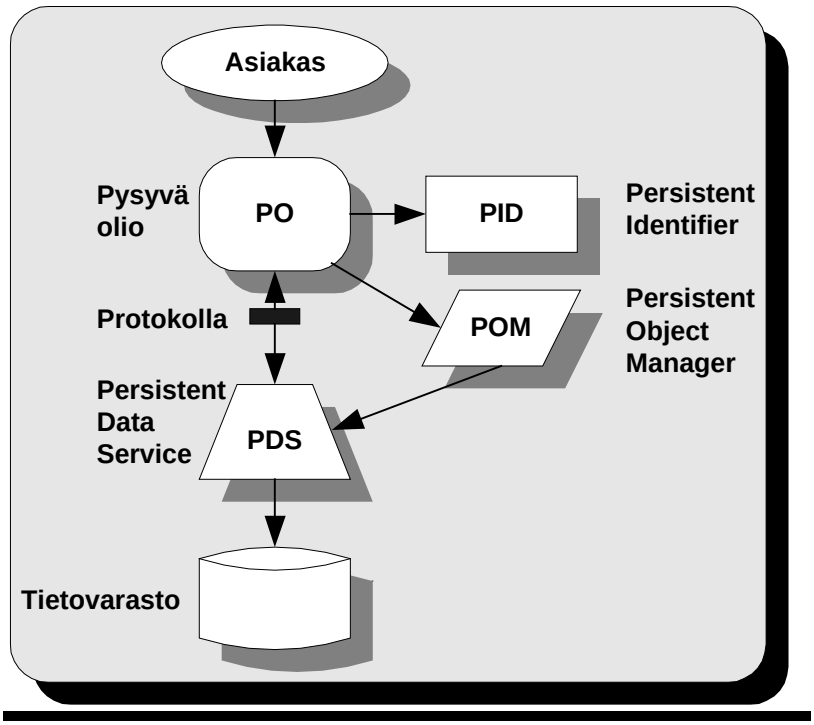
- Transaktio-ominaisuuksilla varustettu palvelin on yhden tai usemman olion muodostama kokoelma, joihin transaktio vaikuttaa mutta joilla ei ole palautettavissa olevia tiloja. Tällainen olio ei osallistu transaktion loppuun saattamiseen, mutta voi virhetilanteessa pakottaa transaktion peruuttamisen. (Orfali et al. 1997, s. 140.)

- Palautettavissa oleva palvelin on samoin yhden tai useamman olion muodostama kokoelma, mutta oloilla on suojattavia transaktioon liittyviä resursseja, kuten tietokantoja, jonoja ja tiedostoja. Oliot rekisteröityvät palvelulle ja lisäksi tarjoavat metodit transaktion

koordinaattorille, joka ohjaa kaksivaiheista transaktion hyväksymisprotokollaa. (Orfali et al. 1997, s. 140.)

Transaktiopalvelu määrittelee neljä toiminnan kannalta olennaista IDL-liityntää (Orfali et al. 1997) mutta ei tietenkään rajoita sitä, kuinka nämä liitynnät tulisi toteuttaa (Baker 1997, s. 398). Olion transaktio-ominaisuus saadaan perimällä TransactionalObject -liityntä. Tämä liityntä on abstrakti luokka, joka ei määrittele operaatioita vaan toimii merkkinä (marker) transaktio-ominaisuuksista. (Orfali et al. 1997, s. .)

Useimmat hajautetut oliot ovat pysyviä poiketen C++-olioista. Tämä tarkoittaa sitä, että olioiden täytyy säilyttää tilansa vielä kauan senkin jälkeen, kun oliot luonut ohjelma päättyy. Ollakseen pysyviä olioiden tila täytyy tallettaa pysyvään tietovarastoon, esimerkiksi tiedostoon tai tietokantaan. Persistent Object Service eli POS mahdollistaa olion pysyvyyden ohittaen olion luoneen sovelluksen tai oliota käyttävän asiakkaan elinkaaren. Olion elinkaari voi olla joko lyhyt tai määrittelemättömän pitkä. POS mahdollistaa olion tilan tallentamisen pysyvään varastoon ja palauttaa olion tilan silloin, kun sitä tarvitaan. Olion sijaitessa paikallisessa muistiavaruudessa sen dataan voidaan viitata niin nopeasti kuin käytettävällä ohjelmointikielellä on mahdollista. POS määrittelee liitynnän dataan ja pysyviin olioihin käyttäen IDL-kielellä määriteltyjä liityntöjä. Näiden liityntöjen toteutukset voivat vaihdella aina kevyistä tiedostojärjestelmistä raskaisiin täysin implementoituihin SQL- tai oliotietokantajärjestelmiin. Tarkoituksena oli luoda avoin toteutus, joka täyttää erilaiset olioiden tallennusvaatimukset. POS määrittelee yhden olioliitynnän, jota käytetään kaikissa tietovarastoissa varaston tyypistä riippumatta. (Orfali et al. 1997, s. 179 - 180.)

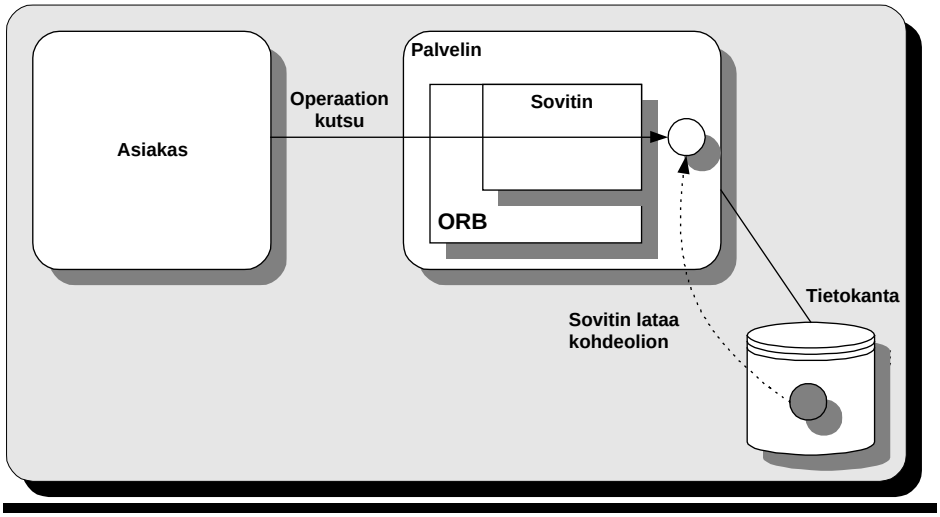


Kuva 14. Pysyvyyspalvelun arkkitehtuuri (Baker 1997, s. 447).

Kuvassa 14 nähdään pysyvyyspalvelun (POS) arkkitehtuuri. Jokaiselle oliolle on annettu PID, joka identifioi olion tilan valitussa tietovarastossa. Jokainen PID sisältää tiedon valitusta tietovarastosta ja oliosta tietovaraston sisällä. PDS (Persistent Data Service) -komponentin tarkoituksena on muodostaa liityntä olion ja siihen sidotun tietovaraston välille sekä hoitaa itse olion pysyvän tilan luku ja/tai talletus. Järjestelmässä voi olla yhtä aikaa useita PDS-komponentteja. Sääntöjä, joilla olio ja PDS kommunikoivat, kutsutaan protokolliksi. PO-palvelun spesifikaatiossa määritellään kolme protokollaa. Standardoidut tai uudet protokollat sisältävät IDL-liitynnän, joka PDS:n tulee toteuttaa, jotta pysyvä olio voi palauttaa ja tallettaa datansa. Ja lisäksi protokollassa määritellään toinen IDL-liityntä, joka pysyvän olion tulee myös toteuttaa, jotta PDS voi lukea ja kirjoittaa olion dataa. Palvelussa määritellään vielä virtuaalimuistin kontrolli PDS:lle. (Baker 1997, s. 447 - 448.)

PO-palvelussa päätöksen siitä, milloin pysyvä tila talletetaan tai palautetaan, voi tehdä joko pysyvää oliota käyttävä asiakas tai itse pysyvä olio. Kun jompikumpi osapuolista haluaa tallentaa tai palauttaa pysyvän tilan, pysyvä olio välittää pyynnön PO-managerille, joka tekee valinnan käytettävästä PDS-komponentista. Tällä kutsun siirtämisellä mahdollistetaan olion siirtymisen käyttämään jotakin toista PDS:ää muuttamatta kuitenkaan omaa koodia tai tilaansa. Tämä tuottaa osan joustavuudesta, jota tarvitaan olion siirtämiseksi tietovarastosta toiseen. On tärkeätä huomata, että mikäli halutaan käyttää toista PDS:ää, tulee pysyvän olion toteuttaa vähintäänkin yksi uuden PDS-komponentin protokolla. (Baker 1997, s. 448.)

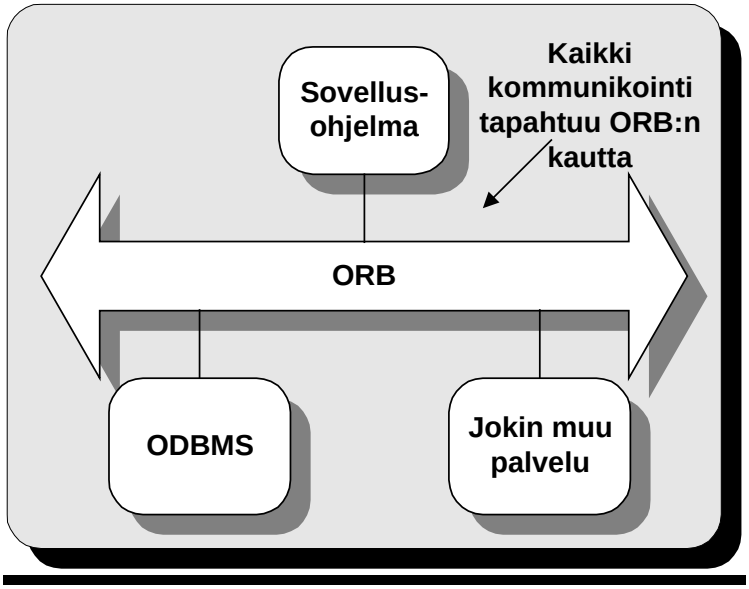
Pysyvyyspalvelu on vain yksi vaihtoehto, jota voidaan käyttää. CORBA-standardi määrittelee toisen lähestymistavan ongelmaan: eräs arkkitehtuurin keskeisistä komponenteista, Basic Object Adapter (BOA), voidaan vaihtaa tai laajentaa, jotta olion pysyvyys saadaan aikaan. BOA on CORBA-ytimen komponentti, joka sijaitsee palvelimen puolella. Se on vastuussa kommunikoinnista ORB:n ja asiakkaiden kutsumien olioiden välillä. BOA:n toiminnallisuuteen kuuluu kutsun kohteena olevan olion paikallistaminen, välitettävien olioviittausten käsittely, lopullisen kutsuttavan operaation tai attribuutin päättely ja lopulta itse kutsun suorittaminen kohdeolioon. Adapteri huomioi, että kohdeolio tai kutsun mukana välitettävien olioviittausten osoittamat oliot voivat sijaita tietokannassa. Tietokantasovitin voi tarvittaessa ladata kohdeolion tietokannasta piilottaen tämän toiminnan kutsun tekevältä asiakkaalta. Kuvassa 15 on esitettynä kaavio toiminnasta. Erona PO-palveluun on se, että tietokantasovitin ei yritä kätkeä tietovarastoa pysyvältä oliolta, vaan pysyvä olio ja tietokanta ovat tiivisti kytkettyjä toisiinsa. Tietokantasovittimen etuina ovat yksinkertaisuus ja mahdollisuus käyttää kaupallisten tietokantatoteutusten tarjoamia erikoisominaisuuksia järjestelmän muuttamisen vaikeutumisen kustannuksella. (Baker 1997, s. 446 - 448.)



Kuva 15. Tietokantasovittimen toiminta (Baker S. 1997, s. 447).

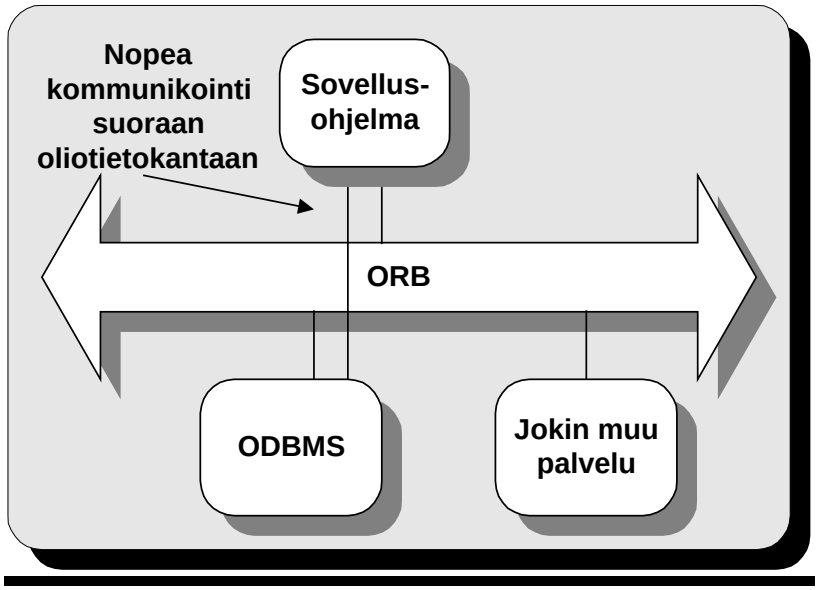
3.4.2 Oliotietokannat ja ORB yhdessä

Kuvassa 16 kaikki kommunikointi palveluiden ja sovellusohjelman olioiden välillä tapahtuu ORB:n kautta. Itse ORB toimii pelkästään välitason ohjelmistona välittäen kommunikoinnin kaikkien osapuolten välillä. Tietokannassa olevat oliot voidaan rekisteröidä järjestelmään CORBA-olioina ja asiakkaat voivat käyttää niitä. Tämän lisäksi sovellusohjelma voidaan kytkeä suoraan oliotietokantaan, kun tarvitaan hyvää suorituskykyä (kuva 17). Tämä ei estä tietokannassa olevien olioiden rekisteröimistä järjestelmään, mikäli muut sovellukset haluavat käyttää näitä olioita kuten tavallisia CORBA-olioita. (Barry 1996, s. 95 - 96.)



Kuva 16. Oliotietokanta yhdistettynä ORB-väylään (Barry 1996, s. 96).

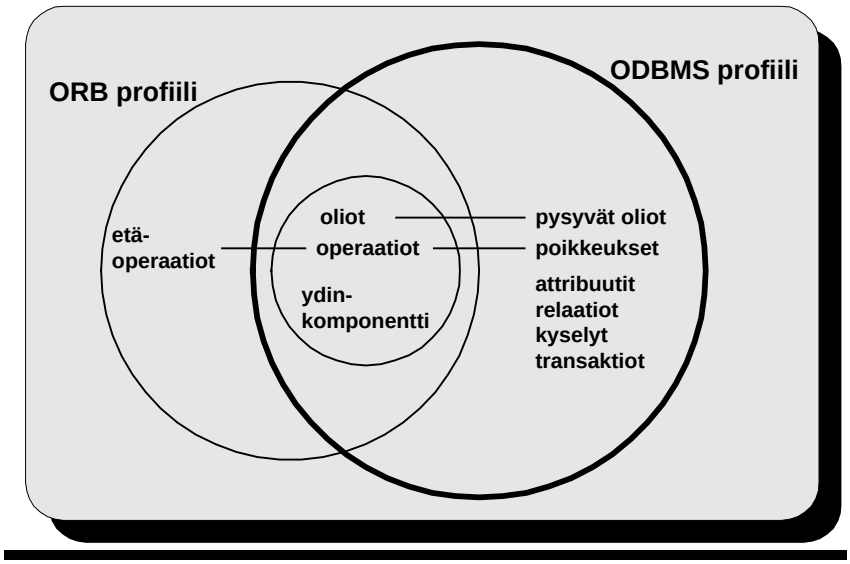
Barryn (1996) mukaan yleinen kysymys on se, voiko ORB korvata palveluillaan oliotietokannan. Kun kaikki palvelut ovat käytettävissä, ORB voi korvata oliotietokannan, jos hidas suorituskyky on hyväksyttävissä. ORB vaatii saman työn tekemiseen useampia järjestelmäkutsuja, mikä vaikuttaa suorituskykyyn. Kun järjestelmän oliot ovat pieniä ja niitä on paljon, oliotietokannat tulevat aina olemaan merkittävästi nopeampia kuin pelkkä ORB. (Barry 1996, s. 96.)



Kuva 17. Oliotietokannan ja sovelluksen suora kytkentä jota sovellusohjelma voi käyttää ohittamalla ORB-väylän (Barry 1996, s. 97).

OMG:n määrittelemä oliomalli on keskittynyt suunnittelun siirrettävyyteen. ODMG:n oliomalli pyrkii lisäksi myös lähdekoodin siirrettävyyteen, keskittyen siirrettävyyteen oliotietokanta järjestelmien teknologia-alueen sisällä. ODMG:n standardikokoelma on suunniteltu mahdollistamaan tietokantasovelluksen kehittäminen käyttäen vain yhtä API-kutsujen joukkoa. (Cattell & Barry 1997, s. 245.)

OMG:n oliomalli jakautuu komponentteihin, joista merkittävin on ydinkomponentti, joka määrittelee luokat ja operaatiot. OMG:n omaksuma teoria on, että jokainen teknologia-alue (GUI, ODBMS jne.) muodostaa profiilin näiden komponenttien joukosta. ORB:n profiiliin kuuluu ydinkomponentin lisäksi tuki etäoperaatioille. Oliotietokannan profiiliin kuuluvat ydinkomponentin lisäksi tuki pysyville olioille, ominaisuuksille (attribuutit ja relaatiot), kyselyille ja transaktioille. OMG:n oliomalli (OMG/OM) on ODMG:n oliomallin (ODMG/OM) osajoukko. Kuvassa 18 on esitettynä eri oliomallien profiilit. (Cattell & Barry 1997, s. 245 - 246.)

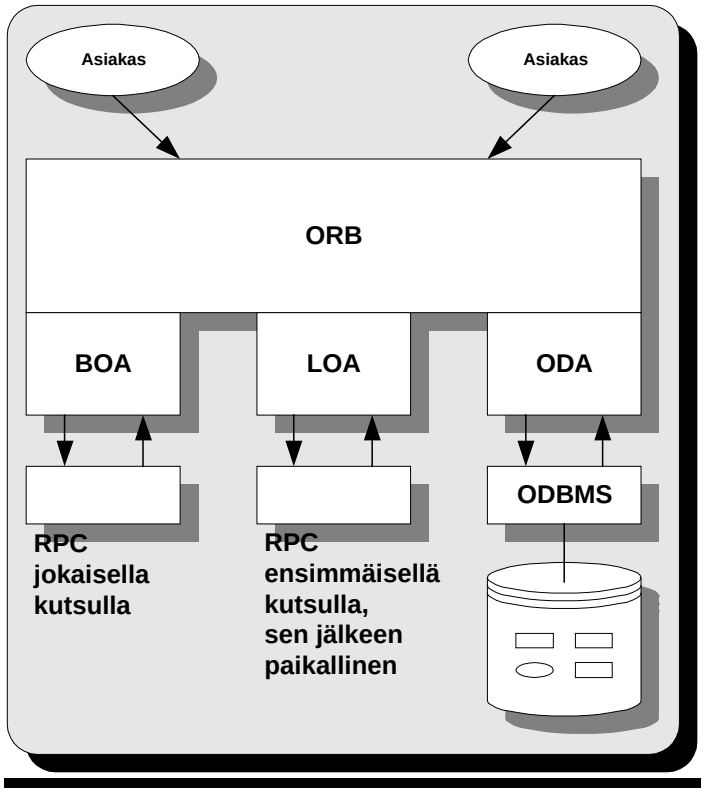


Kuva 18. Oliomallien liittyminen toisiinsa (Cattell & Barry 1997, s. 246).

Cattellin ja Barryn (1997, s. 251) mukaan OMG:n dokumentit eivät vielä käsittele oliotietokannan liittymistä ja toimintaa OMG:n luomassa ORB arkkitehtuurissa. Tärkeitä seikkoja arkkitehtuurien yhdistämisessä ovat (Cattell & Barry 1997, s. 251.)

- suorituskyky, esimerkiksi käyttäen oliota suoraan
- heterogeenisyys ja hajautus sekä hienojakoisten (fine grained) olioiden hallinta oliotietokannalla
- oliotietokannan toiminta oliomanagerina, joka vastaa useista olioista
- ORB:n mahdollisuus käyttää oliotietokantaa järjestelmän yleisenä tietovarastona
- oliotietokannan mahdollisuus käyttää ORB:n palveluja, myös muita järjestelmän olio-tietokantoja.

ODMG esittää oliotietokantasovitinta (ODA) OMG:n määrittelemien sovittimien lisäksi. Se on mekanismi, joka mahdollistaa oliotunnisteiden alijoukon rekisteröinnin järjestelmälle sen sijaan, että kaikki oliotietokannassa sijaitsevat oliot rekisteröitäisiin. Rekisteröidyn alijoukon oliot näyttävät asiakkalle samanlaisilta kuin muutkin ORB:n kautta käytettävissä olevat oliot, joilla on sama olioliityntä. Suorituskyvyn parantamiseksi ODA:n tulisi mahdollistaa olion suora käyttäminen samaan tapaan kuten LOA (library object adapter) oliotietokannan ja ORB:n yhdistävissä sovelluksissa. Kuvassa 19 näkyy oliotietokantasovittimen sijainti järjestelmässä. (Cattell & Barry 1997, s. 254.)



Kuva 19. Oliotietokantasovitin ja sen liittyminen OMG:n arkkitehtuuriin (Cattell & Barry 1997, s. 255).

Ehdotus oliotietokantasovittimen standardoimiseksi tehtiin oliotietokantavalmistajien toimesta ODMG-93-standardissa. ODMG-93 laajentaa pysyvyyspalvelua määritellen tehokkaan PDS-protokollan hienojakoisille olioille. (Orfali et al. 1997, s. 199.)

4. Joustavuusominaisuuksien soveltaminen

Tässä luvussa kerrotaan ECA- ja CORBA-arkkitehtuurien yhdistämisestä sekä yhdistämisessä esiintyneistä ongelmista. Aluksi esitetään, kuinka yhdistäminen tehtiin, ja tämän jälkeen käsitellään tarkemmin ECA-vuorottajan toiminta. Seuraavaksi esitetään vaihtoehtoja toiminnallisuuden muuttamiseen pilottijärjestelmässä ja selvitetään, kuinka pilottijärjestelmää ohjataan käyttöliittymällä. Lopuksi esitetään vaihtoehtoja järjestelmän toimintaa ohjaavien sääntöjen luokkamallin muodostamiseen ja esimerkki sääntöjen muodostamiseksi tuoteominaisuusmalliin perustuen.

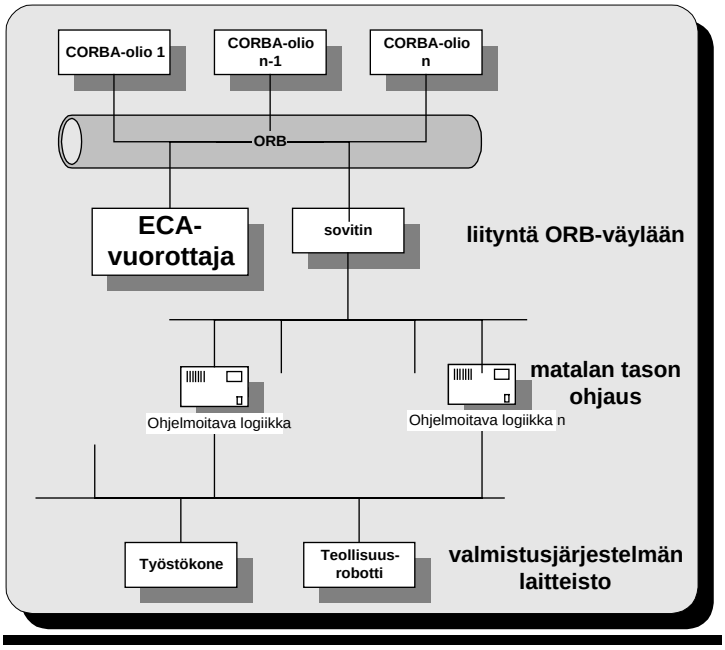
4.1 Liittyminen ohjattavaan järjestelmään

Liittyminen FM-järjestelmään tapahtuu käyttäen tapahtumanvälitykseen ORB-väylää. Tapahtumat välittyvät samaan väylään yhdistetylle ECA-vuorottajalle, joka käsittelee tapahtuman ja suorittaa tapahtumalle määritellyt tehtävät. Kaikki tapahtumat pilot-järjestelmässä ovat ulkoisia ORB-väylään liitettyjen CORBA-olioiden tapahtumia. Tapahtuma aiheutuu CORBA-olion metodin suorituksesta, joko toisen olion kutsuessa tapahtuman lähettävää metodia tai mahdollisesti olion kutsuessa sisäisesti omaa metodiaan. Tapahtumien lähettäminen on eksplisiittistä, eikä tapahdu automaattisesti. Tapahtuma (event) tietorakenteessa välitetään taulukon 5 mukaiset tiedot jotka ovat: tapahtuman lähettäneen olion kantaluokka, olioviittaus lähettäneeseen olioon, sekä metodin nimi, josta tapahtuma lähetettiin.

Taulukko 5. Tapahtumatietorakenteessa välitetyt tiedot.

Kenttä	**Tyyppi**
Olion kantaluokka	merkkijono
Tapahtuman lähettänyt olio	merkkijonona välitetty olioviittaus
Metodin nimi	merkkijono

ECA-vuorottaja toimii välitettyjen tietojen perusteella seuraavasti: tapahtuma käynnistää sääntöön sidotun toiminnallisuuden ja saa aikaan tapahtuman etenemisen edelleen. Tapahtuman propagoituessa esimerkiksi ohjattavaan laitteistoon kytkettyyn komponenttiin saadaan aikaan järjestelmän tilamuutos. Kuvassa 20 on esitettynä liittyminen ohjattavaan valmistusjärjestelmään.

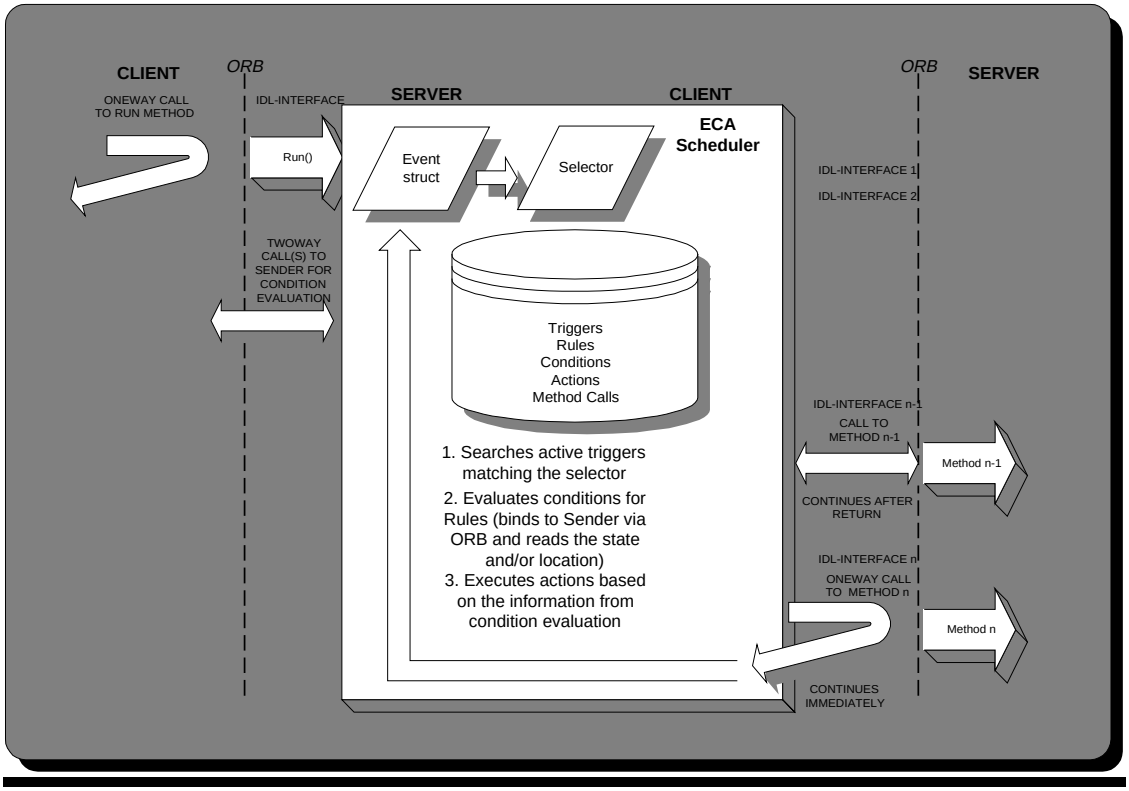


Kuva 20. ECA-vuorottajan liittyminen ohjattavaan järjestelmään.

Sovitin, joka on CORBA-olio, muodostaa rajapinnan ohjattavaan laitteistoon. Tätä liityntää käyttävät muut CORBA-oliot, jotka edustavat ohjattavan laitteiston abstrahoituja komponentteja. ECA-vuorottaja ei itse ohjaa suoraan sovittimen kautta laitteistoa eikä myöskään estä olioiden välistä suoraa kommunikointia. Varsinaisia CORBA-palveluja ei toteutuksessa käytetty. Optimaalisinta olisi, jos valmistusjärjestelmässä käytetyt laitteet tarjoaisivat suoraan CORBA-liitynnän. Jo pelkkä ohjelmoitavien logiikkojen liittyminen suoraan ORB-väylään yksinkertaistaisi kaaviota sovittimen osalta. Estettä laitteiston ohjaamiselle suoraan ECA-vuorottajalla sovitinta käyttäen ei ole.

4.2 Arkkitehtuurimallien yhdistäminen

ECA-vuorottaja näkyy ORB-väylässä muille tavallisena CORBA-oliona tuottaen liitynnän muiden olioiden käyttöön. ORB:n toimiessa ainoastaan tapahtumanvälityskanavana vuorottaja liittyy ORB-väylään pelkästään yhdellä metodilla *Run*, jonka parametrina on tapahtuma-tietorakenne. *Run*-metodi on määritelty yksisuuntaiseksi ja aiheuttaa tapahtuman lähettäneen olion toiminnan jatkumisen välittömästi vuorottajan vastaanotettua tapahtuman. Vuorottaja näkyy palvelimena tapahtuman lähettäneelle asiakkaalle. Tapahtuman vastaanottamisen jälkeen vuorottaja muodostaa välitetystä tapahtumasta sisäiset tapahtuma- ja valitsintietorakenteet. Valitsimen avulla tietokannasta haetaan talletetut liipaisimet, ja näistä päästään suorittamaan loput sääntöön liittyvästä toiminnallisuudesta. Jotta ehto- ja metodikutsuluokat voisivat muodostaa yhteyden ORB-väylässä muihin olioihin vuorottajaan tulee sisällyttää kaikkien kutsuttavien olioiden IDL-käännöksestä saatavat asiakastiedostot. Mikäli liityntöihin tehdään muutoksia, joudutaan korvaamaan muuttuneet tiedostot uusilla ja kääntämään vuorottaja uudestaan uusillaliitynnöillä ORB-väylään. Samoin joudutaan toimimaan myös toisinpäin, mikäli vuorottajan liityntään tehdään muutoksia.

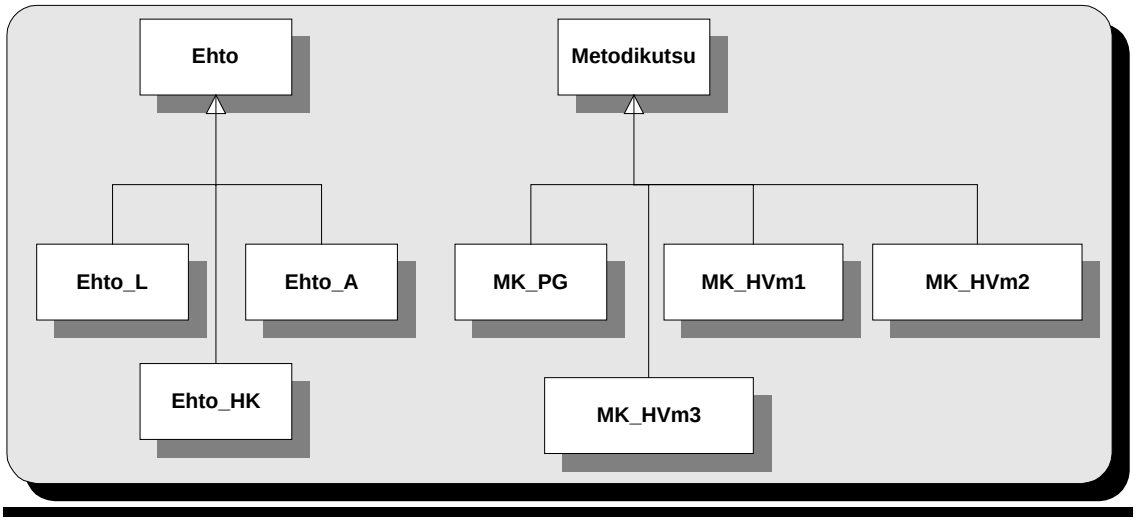


Kuva 21. ECA-vuorottajan toiminta.

Kuvassa 21 näkyy vuorottajan toiminta. Liityntöjä ulospäin tarvitaan niin monta kuin on vuorottajan käyttämiä luokkia. Ehdon evaluointiin käytetyt metodit ovat kaksisuuntaisia, ja johdetuissa metodikutsuluokissa voidaan käyttää joko yksi- tai kaksisuuntaisia kutsuja. Tässä vaiheessa vuorottaja käyttäytyy järjestelmän olioiden liityntöjä käyttävänä asiakkaana. Kun kaikki tapahtumaan sidottu toiminnallisuus on suoritettu, vuorottaja jää odottamaan seuraavaa tapahtumaa, ja seuraava vastaanotettu tapahtuma aloittaa sekvenssin alusta uudelleen.

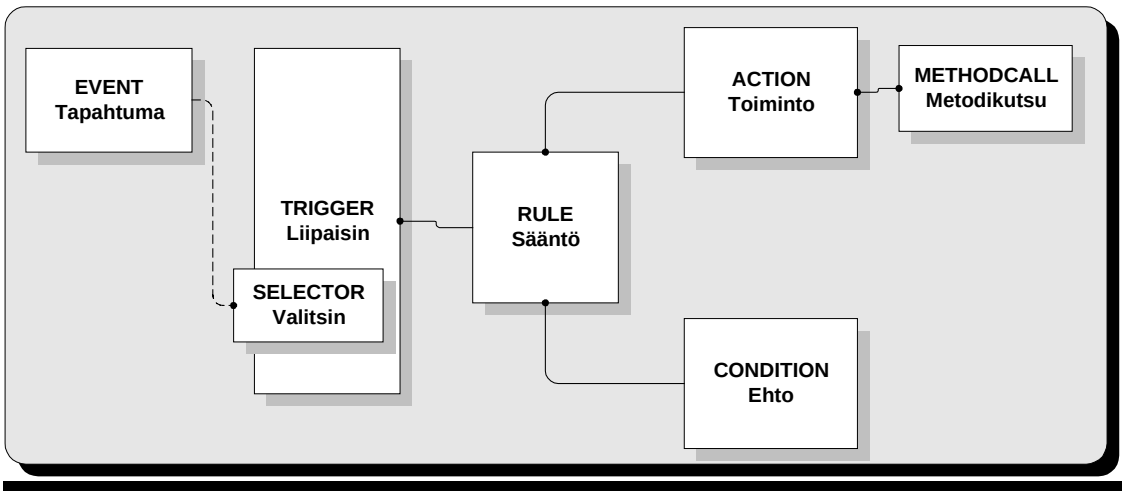
4.3 ECA-konseptin toteutus pilottijärjestelmässä

Järjestelmän toteutuksessa käytetty käännettävä ja vahvasti tyypitetty ohjelmointikieli aiheuttaa vaikeuksia yritettäessä muodostaa yleiskäyttöisiä ehto- ja metodikutsuluokkia. Ohjelmakoodissa käytetyt muuttujatyypit, iteraattorit yms. ovat luokkakohtaisia. Mikäli niitä halutaan käyttää, niiden tulee olla määriteltyinä ohjelman käännösvaiheessa. Kaikki luokkasidonnaiset tietotyypit voisi tietysti koota yhteen luokkaan ja tilanteen mukaan valita niistä sopivat. Aliluokittaminen on toinen vaihtoehto johon päädyttiin.



Kuva 22. Ehto- ja metodikutsuluokkien aliluokittaminen.

Kuvassa 22 nähdään, kuinka ehto- ja metodikutsu luokat on aliluokitettu pilottijärjestelmässä. Aliluokat jakautuivat metodikutsut vastaanottavien luokkien mukaisesti; luokkasidonnaiset tietotyypit jaoteltiin johdettuihin luokkiin. Kantaluokassa ehto on määritelty virtuaalinen metodi *EvaluateCondition*, joka on ylikirjoitettu johdetuissa luokissa. Tässä metodissa suoritetaan ehdon evaluointi. Kantaluokassa on määritelty ehdon evaluoinnille perustoteutus, joka palauttaa aina totuusarvoon TOSI. Metodikutsukantaluokassa on määritelty puhtaasti virtuaalinen metodi *Run*, jolle on kirjotettu toteutukset johdetuissa luokissa. Näihin *Run*-metodien toteutukseen on toteutettuna se toiminnallisuus, mikä kulloinkin tulee suorittaa. Ehto- ja metodikutsuluokat ovat kantaluokkiaan lukuun ottamatta tietokantaan tallennettuja, ORB-väylän kautta ulkoisia olioita käyttäviä asiakkaita.



Kuva 23. ECA-konsepti tietokannassa.

Tietokantaan muodostettu luokkamalli on esitettynä kuvassa 23. Kaikille tietokantaan talletetuille luokille on luotu oma tietokannan koonti -luokan (container) instanssi, johon luodut instanssit talletetaan. Tapahtumaluokkaa ei talleteta yksistään tietokantaan, vaan se tallettuu sääntöä evaluoitaessa toimintoluokan datajäsenenä. Pääasiallisin ero Kappelin et al. (1994) käyttämään malliin on yhdistelmäehtoluokan ja yksinkertaisen ehtoluokan puuttuminen.

Kappelin et al. (1994) toteuttama liipaisimien perintä tapahtuu pilottijärjestelmässä CORBA-liitynnät toteuttavien luokkien perinnän kautta. Mikäli jokin perittävän luokan metodeista on määritelty liipaisevaksi, tämä toiminnallisuus on myös johdetuilla luokilla. Mikäli aktiivista toiminnallisuutta johdetussa luokassa halutaan muuttaa, tulee kyseinen metodi määritellä kantaluokassa virtuaaliseksi olio-ohjelmoinnin periaatteiden mukaisesti.

4.3.1 ECA-vuorottajan toiminta tarkemmin

ECA-vuorottajan toiminta käynnistyy ORB:n välitettyä sille tapahtumatietorakenne. *Run*-metodin suoritus alkaa transaktion aloittamisella, minkä jälkeen muodostetaan välitetyistä tiedoista sisäinen tapahtumatietorakenne, jonka kenttiin sijoitetaan välitetyt arvot. Merkkijonona lähetetystä olioviittauksesta erotetaan lähettäneen olion nimi, joka lisätään sisäiseen tapahtuma tietorakenteeseen. Tämän jälkeen täytetään liipaisimien hakua varten valitsimen kentät. Säännön suorittamisessa käytetyt dynaamiset listat tyhjennetään, ja valitsimen kentistä muodostetaan kyselyssä käytettävä predikaattimerkkijono. Iteraattoria käyttäen tietokannasta haetaan aktiiviset liipaisimet ja sellaiset liipaisimet, joiden valitsin vastaa predikaatissa annettuja arvoja. Iteraattorin sisältämät tapahtumasta ‘kiinnostuneet’ liipaisimet käydään läpi silmukassa. Jokaisesta liipaisimesta tarkistetaan oliokohtainen deaktivointi. Mikäli deaktivointi ei ollut asetettuna, liipaisimeen sidottuun sääntöluokan instanssiin navigoidaan assosiaatiota käyttäen. Sääntö lisätään evaluoitavien sääntöjen dynaamiseen listaan. Iteraattorin läpikäynnin jälkeen kaikki evaluoitavat säännöt ovat listassa.

Evaluoitavien sääntöjen lista käydään läpi silmukassa. Ensiksi kutsutaan säännön *EvaluateRule*-metodia. Tämä kutsu selvittää assosiaatiota käyttäen sääntöön sidotun ehto-osan ja kutsuu sen *EvaluateCondition*-metodia. Ehto evaluoidaan, ja mikäli ECA-vuorottajalle palautuu ehdon suorituksen tuloksena totuusarvo tosi, kutsutaan sääntöön assosioidun toiminto-osan *UpdateAction*-metodia, joka tallettaa ehdon evaluoinnista saadun tiedon toiminto-osaan. Lopuksi sääntö lisätään viivästettyjen sääntöjen listaan prioriteettinsa mukaiseen paikkaan. Tässä käytetään apuna MFC-luokkakirjaston perittyä CList-luokkaa, johon on lisätty prioriteetin huomioiva lisäysmetodi.

Kaikki säännöt, joiden toiminto-osa pitää suorittaa, ovat nyt dynaamisessa listassa. Tämä lista käydään läpi silmukassa. Toiminto-osan suorittaminen tapahtuu kutsumalla säännön *ExecuteAction*-metodia, joka kutsuu sääntöön assosioidun toimintoluokan instanssin *ExecuteSequence*-metodia. Tämä metodi käy läpi toiminto-osaan assosioidut metodikutsuluokkien instanssit ja suorittaa lopulta niiden *Run*-metodit välittäen niille aiemmin toiminto-osaan talletetut ehdon evaluoinnin tulokset.

Ehdon evaluointi ORB-väylän kautta alkaa vuorottajan yhteyden muodostamisella tapahtuman lähettäneeseen CORBA-olioon, kutsuen sen jälkeen olion *lueTila*- ja/tai *lueSijainti*-metodia. Saatu tieto lisätään sisäiseen tapahtumatietorakenteeseen (taulukko 6), johon on lisätty kentät luetulle tiedolle. Ehdosta riippuen näiden kahden kentän kombinaatiot ovat käytössä. Kenttien asettamisen jälkeen ehto-osa palauttaa boolean-tyyppisen tiedon säännön suorituksen jatkamisesta. Tapahtumatietorakenne välittyy metodikutsu-osalle toiminto-osaa suoritettaessa, ja mikäli samaan johdettuun toiminto-osan luokkaan on toteutettuna useampi toiminnallisuus, suoritetaan välitetyn tiedon perusteella haaraantuminen oikeaan paikkaan koodissa.

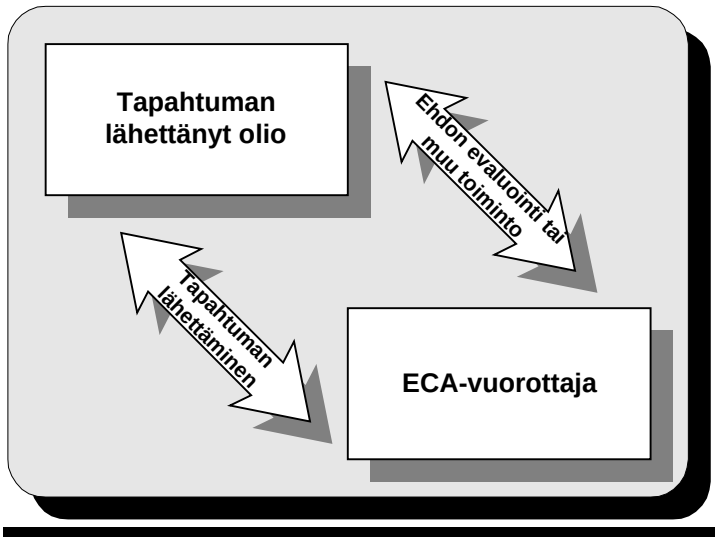
Taulukko 6. ECA-vuorottajan käyttämä sisäinen tapahtumatietorakenne.

Kenttä	**Tyyppi**
Olion kantaluokka	merkkijono
Tapahtuman lähettänyt olio	merkkijonona välitetty olioviittaus
Metodin nimi	merkkijono
Olion nimi	erotetaan olioviittauksesta (merkkijono)
Tila	ehdon evaluoinnissa luettu CORBA-olion tilatieto (merkkijono)
Sijainti	ehdon evaluoinnissa luettu CORBA-olion sijaintitieto (merkkijono)

Jotta johdettujen ehto-luokkien määrän kasvamiselta vältyttiin, yhdessä tapauksessa käytettiin olion liitynnän laventamista (widening) (Baker 1997, s. 100). Sidonta tapahtuu lähettäneen olion kantaluokkaan. Ehdon evaluoinnissa voidaan tuolloin käyttää vain sellaisia metodeja, jotka ovat kantaluokassa.

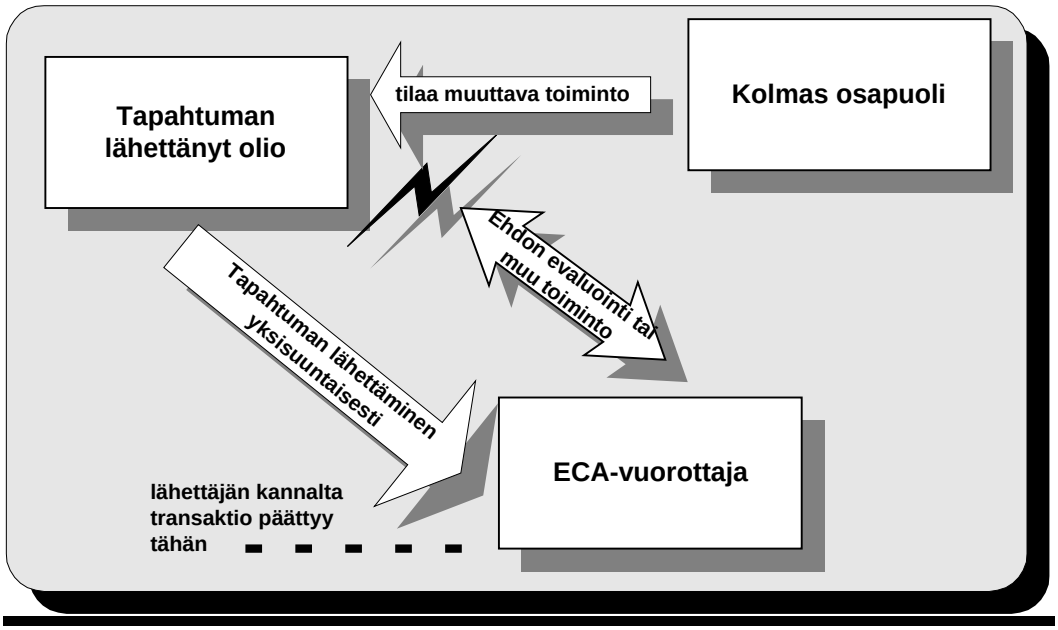
4.3.2 Vuorottajan toteutuksessa esiintyneitä ongelmia

Tapahtuman välityksessä lähettäjältä ECA-vuorottajalle joudutaan käyttämään yksisuuntaista (oneway) kutsua. Mikäli kaksisuuntaista kutsua käytetään, lähettäjä jää odottamaan paluuarvoa ECA-vuorottajalta kutsumaansa *Run*-metodiin. Tapahtuman lähettänyt olio ei tuolloin palvele sille tulevia metodikutsuja. Mikäli ECA-vuorottaja yrittää suorittaa tapahtuman lähettäneen olion metodeja joko ehtoa evaluoitaessa tai suorittaessa metodikutsuluokan *Run*-metodia, oliot jäävät odottamaan toisiaan. Syntyy kuvan 24 mukainen dead-lock-tilanne.



Kuva 24. Tapahtuman lähettänyt olio jää odottamaan vastausta eikä palvele vuorottajalta tulevaa metodikutsua.

Yksisuuntaisen kutsun käyttäminen johtaa kuitenkin toiseen ongelmaan. Transaktion rajat rikkoontuvat; tapahtuman lähettämisen jälkeen ennen vuorottajan suorittamia operaatioita jää aikajakso, jossa muut järjestelmän oliot voivat suorittaa tapahtuman lähettäneeseen olioon sen tilaa muuttavia operaatioita. Yksi tai useampia tiloja menetetään. Tämä on esitettynä kuvassa 25.



Kuva 25. Yksisuuntaisen kutsun käyttäminen tapahtuman välittämisessä aiheuttaa ongelmia. Vuorottaja voi lukea väärän tilatiedon.

Dayal et al. (1988) esittämän mallin mukaisesti ehdot ovat tietokantaan suoritettavia kyselyjä. Ehdon evaluointia ei kuitenkaan voitu toteuttaa tällä tavalla pilot-järjestelmässä. Jotta tila voitaisiin lukea tietokannasta, tulee ohjattavan järjestelmän entiteettejä kuvaavien CORBA-olioiden tila ‘peilata’ tietokantaan. Koska käytössä ei ollut CORBA-POS-palvelua tai toimivaa tietokantasovitinta, ehdon evaluointi jouduttiin suorittamaan ORB-väylän kautta metodikutsuna. Yleiskäyttöisen tyypittömän tietokantaan kohdistuvan kyselyn tekeminen osoittautui vaikeaksi tehtäväksi, eikä sitä saatu luotettavasti toimimaan. Tätä ehtotyyppiä (ehdon kantaluokkaa) käytetään kuitenkin sellaisissa säännöissä, jotka tarvitsevat aina totuusarvon tosi.

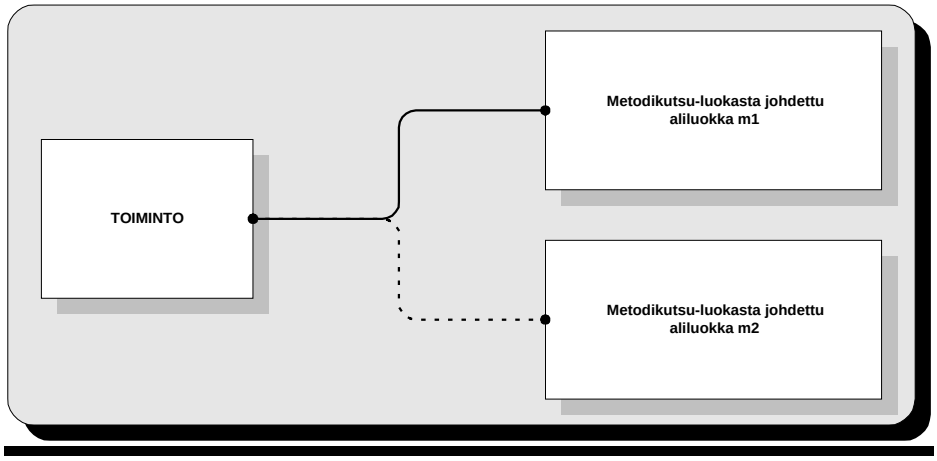
Tapahtuman välityksessä käytettyä yksisuuntaista kutsua ei voi pitää luotettavana menetelmänä. Tapahtumien nopea lähettäminen kuvan 27 käyttöliittymältä johtaa TCP/IP-socket liitynnän virheilmoitukseen, joka ilmoittaa samanaikaisen operaation olevan jo suorituksessa.

4.3.3 Toiminnallisuuden muuttaminen

ECA-konseptin käyttäminen tarjoaa useita mahdollisuuksia järjestelmän toiminnallisuuden muuttamiseen. Pilottijärjestelmässä toiminnallisuuden muuttaminen rajoittuu seuraaviin vaihtoehtoihin:

- liipaisimen aktivointi tai deaktivointi kokonaan tai oliokohtainen deaktivointi. Tämä on osa liipaisimen perustoiminnallisuutta
- assosiaatioiden muuttaminen tietokantaan tallennettujen instanssien välillä. Tämän mahdollistaa käytetty tietokanta
- kokonaan uuden sääntöketjun luominen tai vastaavasti olemassa olevan sääntöketjun poistaminen
- kokonaan uuden sääntökannan toimittaminen.

Kuvassa 26 nähdään järjestelmän toiminnallisuuden muuttaminen vaihtamalla olioiden välistä kytkentää eli assosiaatiota. Mikäli järjestelmän toimintaa halutaan muuttaa esitetyllä tavalla, on toiminnalle määriteltävä vaihtoehtoinen toteutus. Tämä tarkoittaa uuden aliluokan määrittelemistä metodikutsuluokasta, sen Run-metodin toteutusta ja lopulta instanssin tekemistä uudesta aliluokasta sääntökantaan.



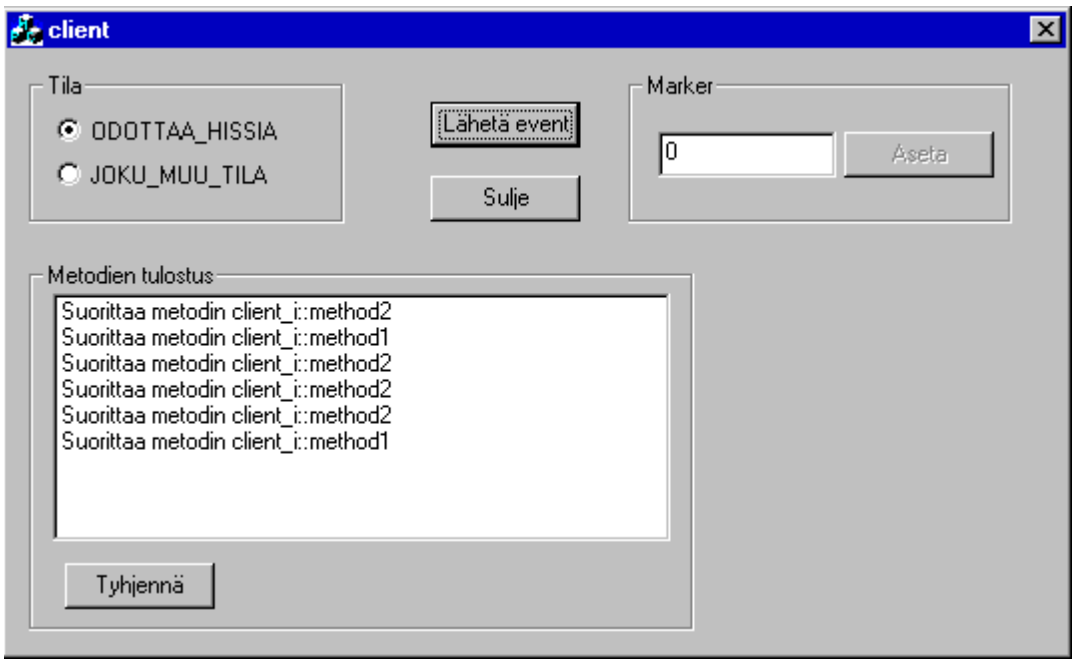
Kuva 26. Toiminnallisuuden muuttaminen assosiaatiota vaihtamalla.

Koska pilottijärjestelmässä oli tarkoituksena toteuttaa järjestelmän toimintaa ohjaava perusohjelmisto, ei toiminnallisuuden muuttamista päästy kokeilemaan. Kuitenkin edellä mainittujen vaihtoehtojen toimivuus todennettiin koejärjestelyllä, jossa toteutettiin

- asiakas (kuva 27), joka lähettää tapahtumia ECA-vuorottajalle toimien vuorottajan asiakkaana. Asiakas toteuttaa liitynnän, joka tarjoaa kolme metodia ORB-väylään. Yksi metodeista on ehdon evaluointia varten ja kaksi muuta visualisoivat ainoastaan toiminnallisuuden muuttumista tulostaen kaksi eri merkkijonoa.

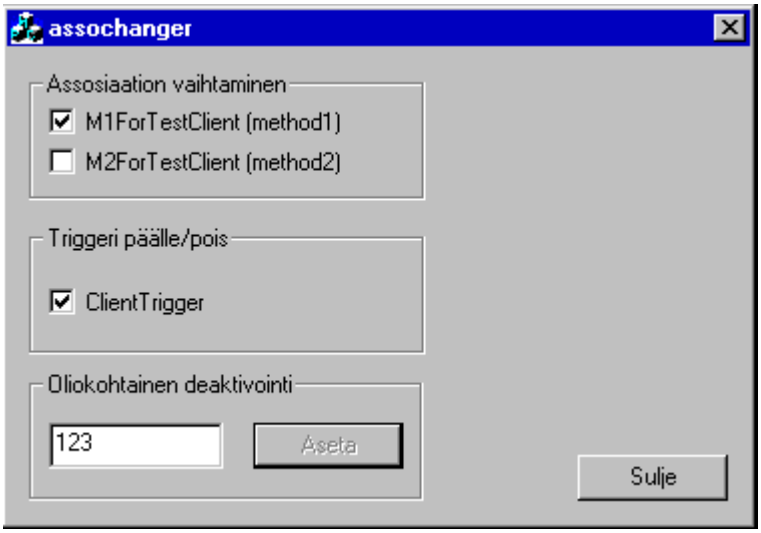
- assosiaatiota muuttava ohjelma, joka kuvassa 26 esitetyn toiminnan lisäksi käsittelee liipaisimen datajäseniä.

Metodikutsuluokasta johdettiin kaksi aliluokkaa, joista toinen kutsuu asiakkaan metodia *method1* ja toinen metodia *method2*. Lisäksi ehtoluokasta johdettiin aliluokka, joka tarkistaa asiakkaan tilatiedon käyttäen asiakkaan metodia *lueTila*.



Kuva 27. ECA-vuorottajalle tapahtumia lähettävä asiakas.

Kuvassa 27 näkyy asiakkaan toiminnan muuttuminen. Riippuen assosiaatioiden tilasta tulostus muuttuu sen mukaisesti. Metodeista yksi tai kaksi suoritetaan joko molemmat, ainoastaan toinen tai ei kumpaakaan. Kuvassa 28 on esitettynä assosiaatiomuutokset tekevän ohjelman käyttöliittymä. Ohjelma muodostaa primitiivisen käyttöliittymän sääntökannan yhteen sääntöön.

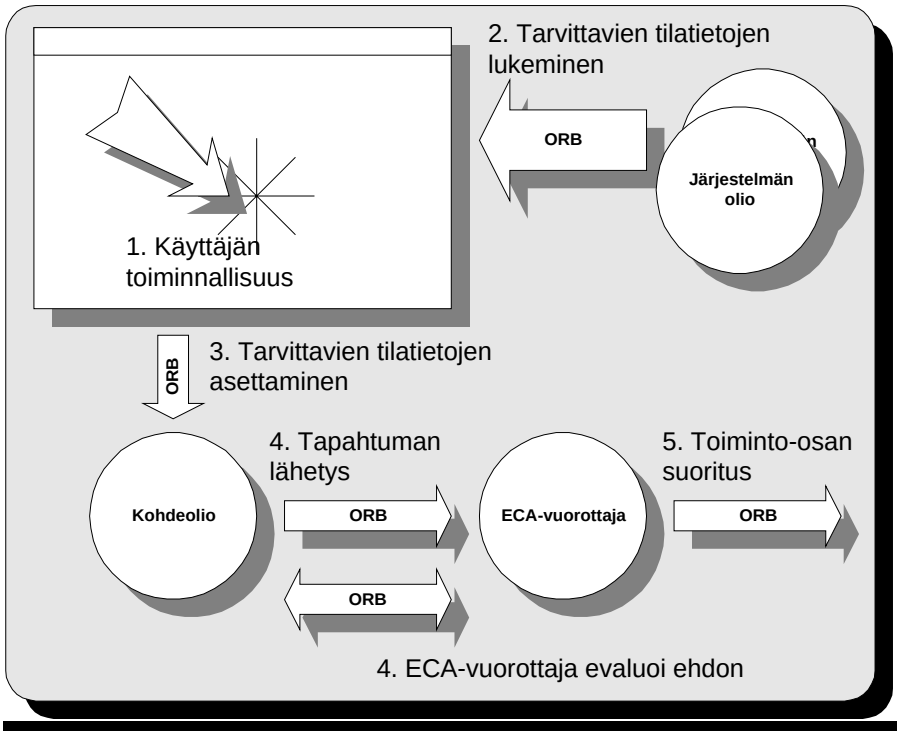


Kuva 28. Assosiaatiomuutokset tekevä ohjelma.

Sääntökantatiedoston vaihtaminen vaatii tietokantaa käyttävien sovellusten uudelleen käynnistämisen. Tämän jälkeen järjestelmän toiminta muuttuu uutta sääntökantaa vastaavaksi.

4.4 Käyttöliittymän ja ECA-konseptin yhdistäminen

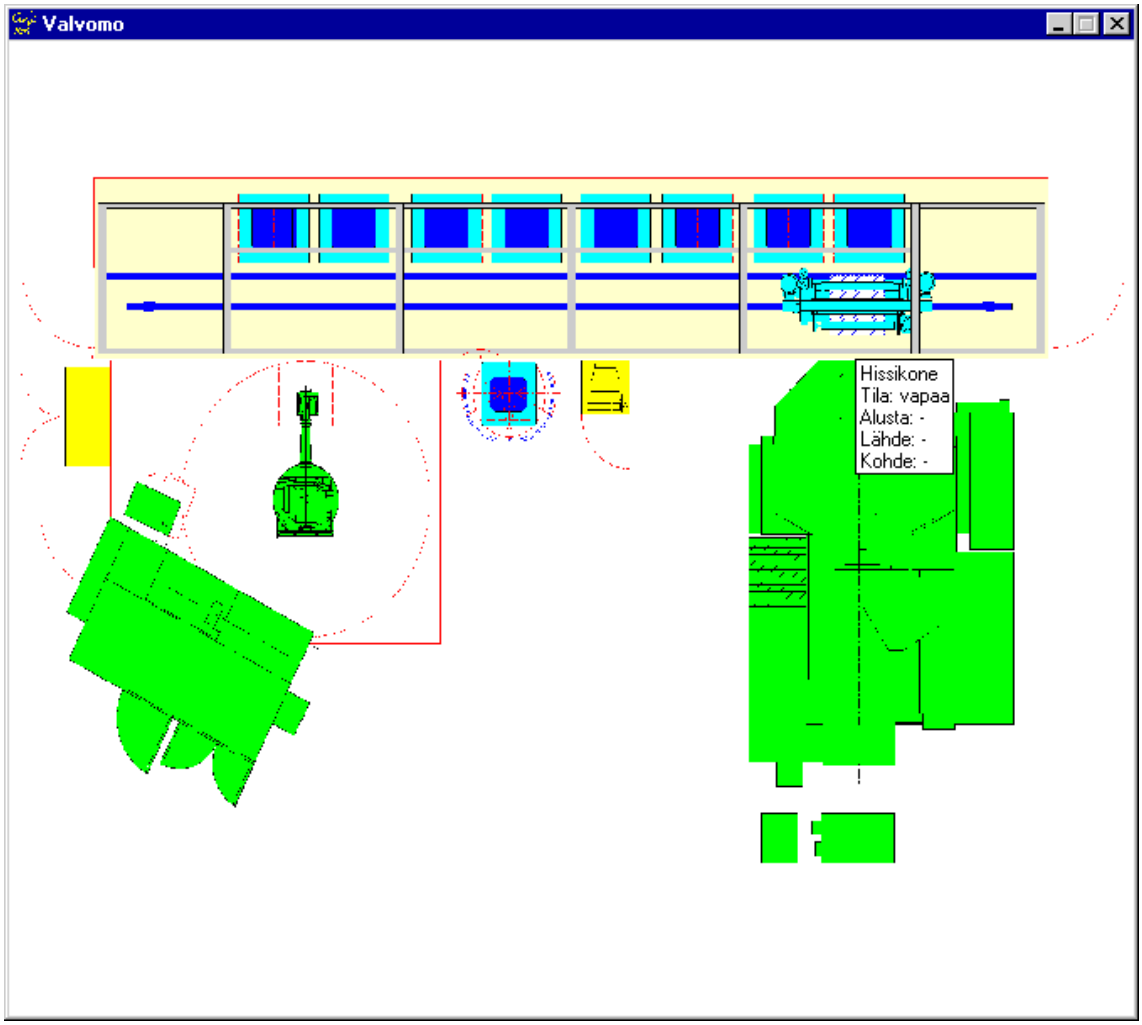
Järjestelmän ohjaus liittyy ORB-väylään samoin kuin muutkin järjestelmän komponentit. Ohjaus suoritetaan graafisella Windows-käyttöliittymällä, joka toimii ohjattavan järjestelmän valvomo näyttönä. Käyttöliittymään on sisällytetty CORBA-luokka, joka toimii pelkästään muita järjestelmän olioita käyttävänä asiakkaana. Järjestelmä aktivoituu käyttäjän suorittaessa joitakin toimenpiteitä valvomonäytöllä, esimerkiksi lavan siirtämisen työstökoneelle. Valvomonäyttö suorittaa toiminnallisuuten tarvittavat operaatiot, kuten toimintaan liittyvien järjestelmän olioiden tilan tarkistuksen. Mikäli toiminta oli hyväksyttävissä, valvomonäyttö muuttaa kohdeolion tilaa koodinsa mukaan. Kun jokin tilatietojen asettamiseen liityvä metodi on määritelty tapahtuman lähettäväksi metodiksi, tapahtuman suoritus siirtyy ECA-vuorottajalle ja siitä edelleen eteenpäin. Kuvassa 29 toiminnallisuus on esitettynä yleisellä tasolla.



Kuva 29. Järjestelmän toiminnan aktivointi käyttöliittymästä.

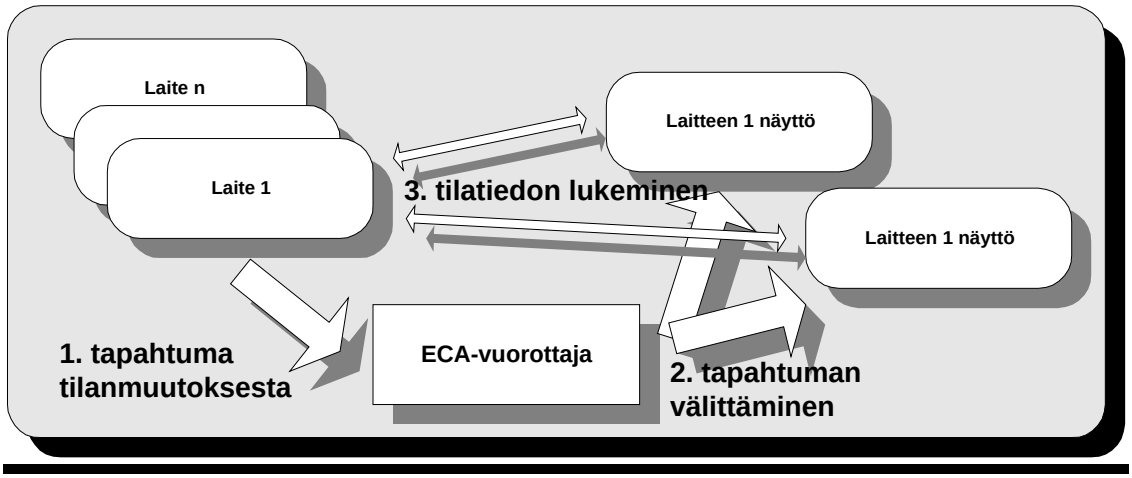
Valvomosta muodostuu järjestelmän muihin CORBA-olioihin sidottu osa, ja ohjattaviin komponentteihin tehtävät muutokset heijastuvat näin ollen myös valvomonäyttöön. Valvomonäytön toteuttajan tulee tuntea järjestelmän toiminallisuus tarkkaan ja tietää, mitä seurauksia jonkin tapahtuman lähettävän metodin kutsumisella on. Koska valvomonäyttö on pelkästään järjestelmän muita olioita käyttävä asiakas, järjestelmän oliot eivät voi lähettää tietoa suoraan valvomonäytölle esimerkiksi tilojensa muutoksista. Tilatiedon lukeminen ja näyttäminen ratkaistiin dynaamisesti muodostettavilla työkaluvihjeillä (tool tip), ja tilatieto luetaan ORB-väylää käyttäen. Kuvassa 30 käyttäjä on vienyt kohdistimen järjestelmän hissikoneolion päälle, olion tilatieto on luettu ja

tulostettu. ORB-väylää hyödynnettiin myös valmistusjärjestelmän varaston muodostamisessa. Varaston koko- ja sisältötiedot luetaan niitä ylläpitävältä varasto-oliolta, ja varastonäyttö muodostetaan näiden tietojen perusteella dynaamisesti.



Kuva 30. Pilottijärjestelmän valvomonäyttö.

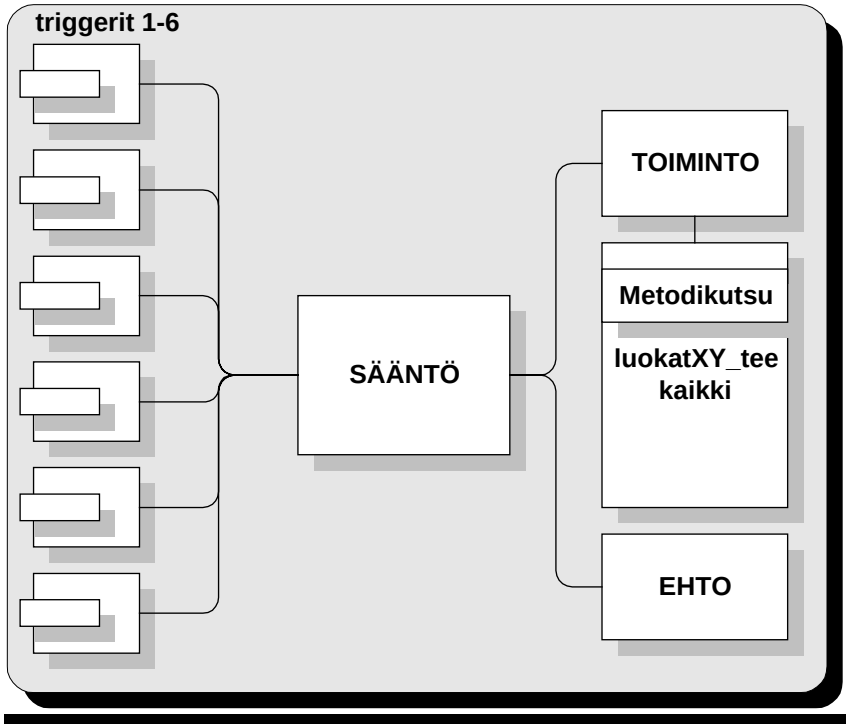
ECA-vuorottajaa käytettiin myös tapahtumien välittämiseen järjestelmän laitteiden ja niiden tilaa seuraavien käyttöliittymien välillä. ECA-vuorottaja ylläpitää tietoja järjestelmään instantioiduista laitteista ja niitä seuraavista käyttöliittymistä. Laitteen muuttaessa tilaansa tapahtuma välittyy ECA-vuorottajalle. Sääntöön sidotussa toiminto-osassa tämä tapahtuma välitetään kaikille kyseistä laitetta projisoiville käyttöliittymille. Koska tapahtuma-tietorakenteesta saadaan selville ainoastaan tapahtuman lähettäneen olion nimi, sen kantaluokka ja lähettänyt metodi, joudutaan laitteen näytössä lopulta suorittamaan tilatiedon kysely laitteelta. Laitetta ja sen käyttöliittymää ei saada täysin erotettua toisistaan. Tieto siitä, mitä laitetta kukin käyttöliittymä seuraa, tallentuu kahteen paikkaan, ECA-vuorottajaan ja itse käyttöliittymään. Kuva 31 selventää tilannetta.



Kuva 31. Tapahtumatiedon puutteellisuudesta johtuva takaisinkytkentä.

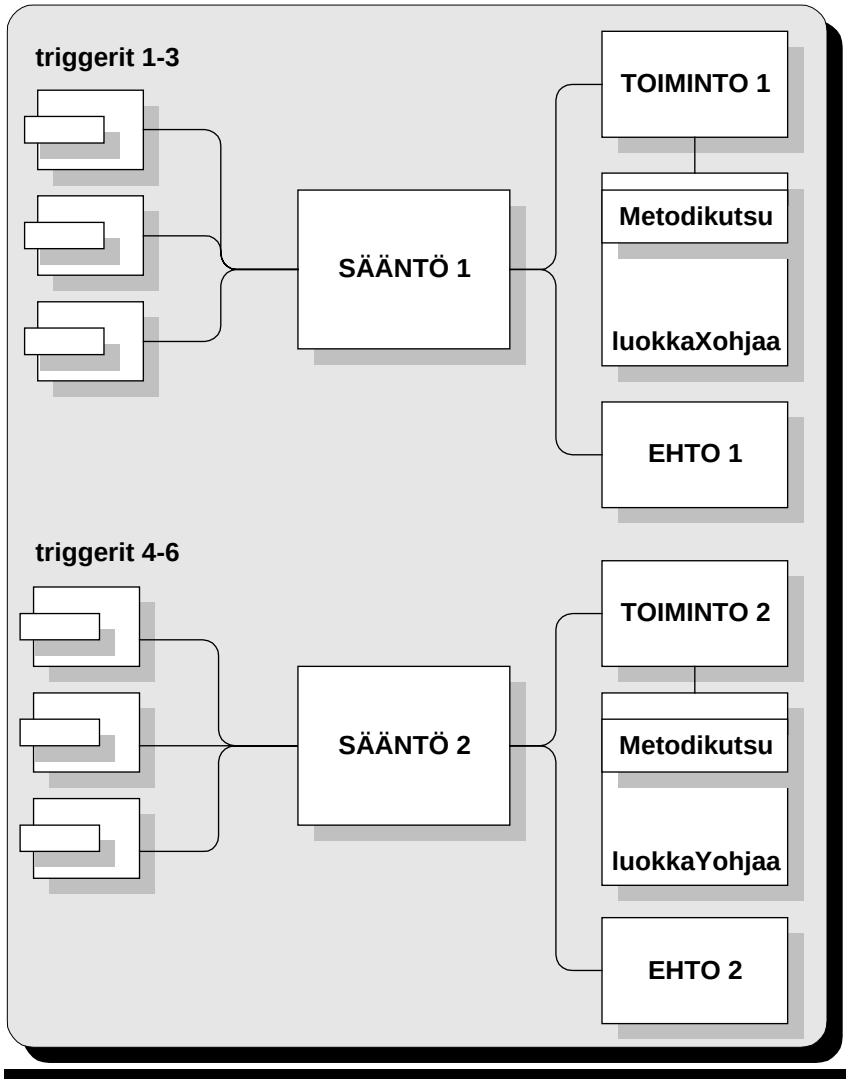
4.5 Vaihtoehtoja ECA-sääntöjen luokkamallin suunnitteluun

Ensimmäinen malli säännön muodostamiselle on esitetty kuvassa 32. Kuusi eri tapahtumaa, jotka liittyvät kahteen eri luokkaan, on assosioitu samaan sääntöön. Tämä sääntö liittyy toiminto-osaan, johon on kytketty ainoastaan yksi metodikutsuluokasta johdettu aliluokka. Koska tieto tapahtumasta välittyy metodikutsuluokalle, voidaan tämän perusteella haaraantua oikeaan suoritettavaan operaatioon yksinkertaisella if-rakenteella tai esimerkiksi switch-case-rakennetta käyttäen. Tätä mallia voidaan käyttää ainoastaan silloin, kun kaikilla tapahtumilla on sama ehto-osa, esimerkiksi aina tosi -ehto. Joustavuus rajoittuu liipaisimien käsittelyyn ja ehto-osan vaihtamiseen. Metodikutsuluokan *Run*-metodista muodostuu helposti pitkä ja vaikeaselkoinen. Tietokantaan talletettavien luokkien määrä pysyy kuitenkin pienenä.



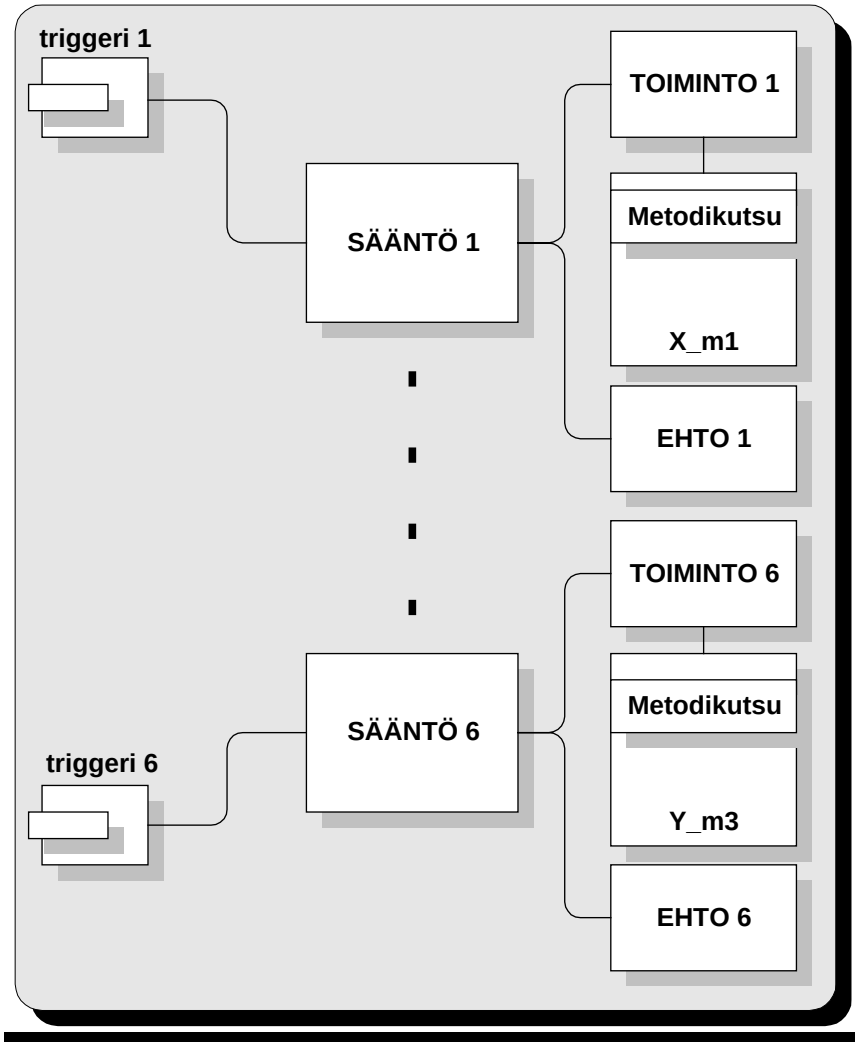
Kuva 32. Kuuden eri tapahtuman ja kahden eri luokan vaatima toiminnallisuus toteutettuna yhdellä metodikutsuluokan johdetulla luokalla.

Toinen vaihtoehto (kuva 33) jakaa toiminnallisuutta vastaanottavien luokkien kesken muodostamalla metodikutsuluokasta kaksi aliluokkaa. Ehto-osia ei tarvitse välttämättä sitoa enää samaan ehtoon. Edelleen joudutaan kuitenkin haaraantumaan metodikutsuosassa tapahtumatiedon perusteella oikeaan tapahtumankäsittelijään. Joustavuus laajenee kuitenkin eri ehto- ja sääntöosien käytön myötä. Säännöt voidaan tarvittaessa priorisoida, ja mikäli sääntöjen eri kytkentätavat on toteutettu ECA-vuorottajassa, saadaan säännöt suoritettua kytkentätavan mukaisesti.



Kuva 33. Sääntökanta, kun luokkien toiminnallisuus on hajautettu kahteen eri Metodikutsu-luokan aliluokkaan.

Kolmas vaihtoehto jakaa toiminnallisuuden pienimpiin mahdollisiin kokonaisuuksiin. Jokaiselle liipaisimelle muodostuu oma metodikutsualiluokka, ja samoin voidaan tarvittaessa jokainen liipaisin sitoa omaan sääntöönsä. Instansseja ja johdettuja aliluokkia muodostuu paljon, pahimmassa tapauksessa jokaisesta ehto- ja metodikutsuluokasta omansa, eli tässä tapauksessa 12 johdettua luokkaa. Tämä vaihtoehto on esitetty kuvassa 34. Muutosten tekeminen on kohdistettavissa selkeästi tiettyihin luokkiin, jotta vaihtoehtoja toiminnallisuuden muuttamiseen on eniten.



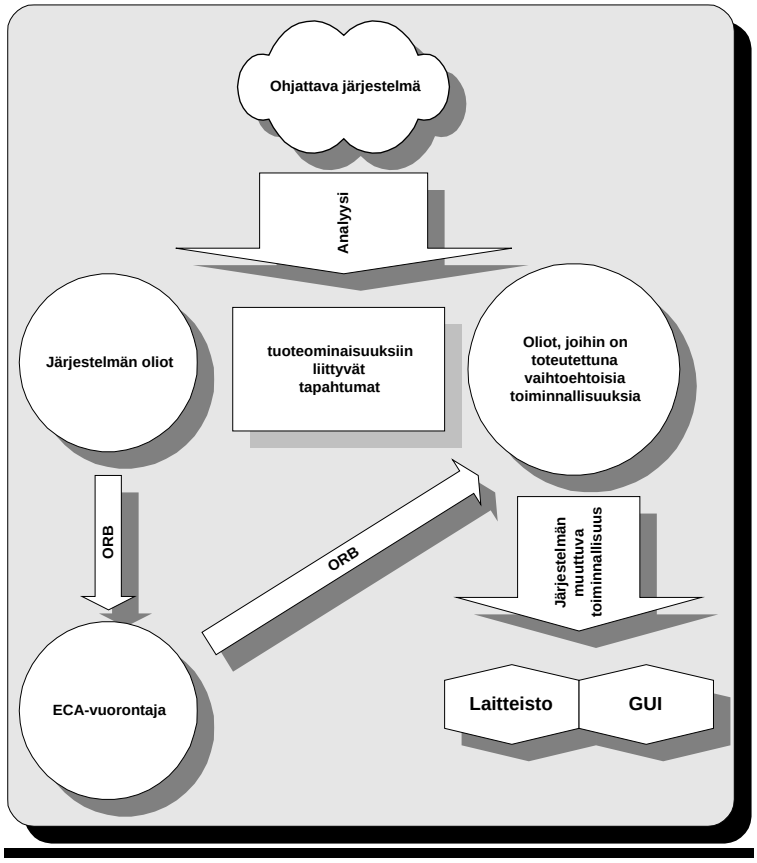
Kuva 34. Sääntökanta, kun haluttu toiminnallisuus on hajautettu mahdollisimman pieniin kokonaisuuksiin.

Esitetyt vaihtoehdot eivät tietenkään ole ainoat mahdolliset. Eri malleja voidaan soveltaa samassa sääntökannassa tarpeen mukaan, ja lisäksi voidaan yhdistellä tietokantaan luotuja instansseja ja jakaa niitä sääntöjen kesken. Yksi mallin mahdollistama vaihtoehto on liittää useampi kuin yksi metodikutsualiluokka toiminto-osaan. On toki huomattava, että myös ehto-osissa toiminnallisuutta voidaan koota yhteen samoin kuin metodikutsuluokissa. Tämä lisää entisestään vaihtoehtojen määrää. Selkeää suunnittelumallia sääntökannan muodostamiseksi tehdyn työn pohjalta ei voi esittää.

4.5.1 Tuoteominaisuuksiin perustuva järjestelmän kehittäminen

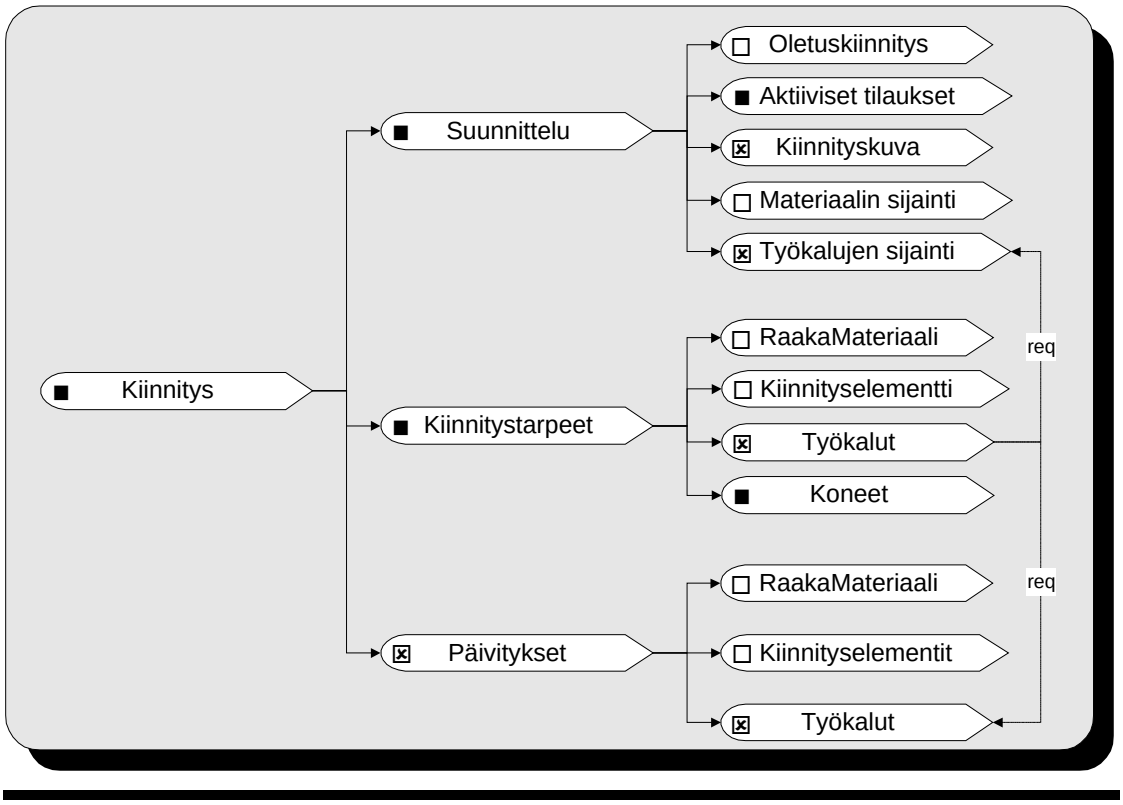
Kehitetty perusohjelmisto mahdollistaa teoriassa yksinkertaisen tuoteominaisuuksien asiakas kohtaisen asettelun. Tämä vaatii tuoteominaisuuksien tunnistamista järjestelmästä, koska käytännössä kaikki tuoteominaisuudet joudutaan toteuttamaan järjestelmän mukana toimitettaviin

komponentteihin. ECA-vuorottaja toimii pelkästään tuoteominaisuuksien kytkimenä. Kuvassa 35 on ohjattavan järjestelmän analyysin perusteella tunnistettu järjestelmän oliot, tapahtumat, näytöt ja toiminallisuuden jakautuminen eri olioihin. Näiden tietojen perusteella voidaan muodostaa sääntökanta, jota muuttamalla saadaan aikaan toiminnallisuuden asiakaskohtainen muuttaminen.



Kuva 35. Esimerkki järjestelmän kehittämisestä tuoteominaisuuksiin perustuen.

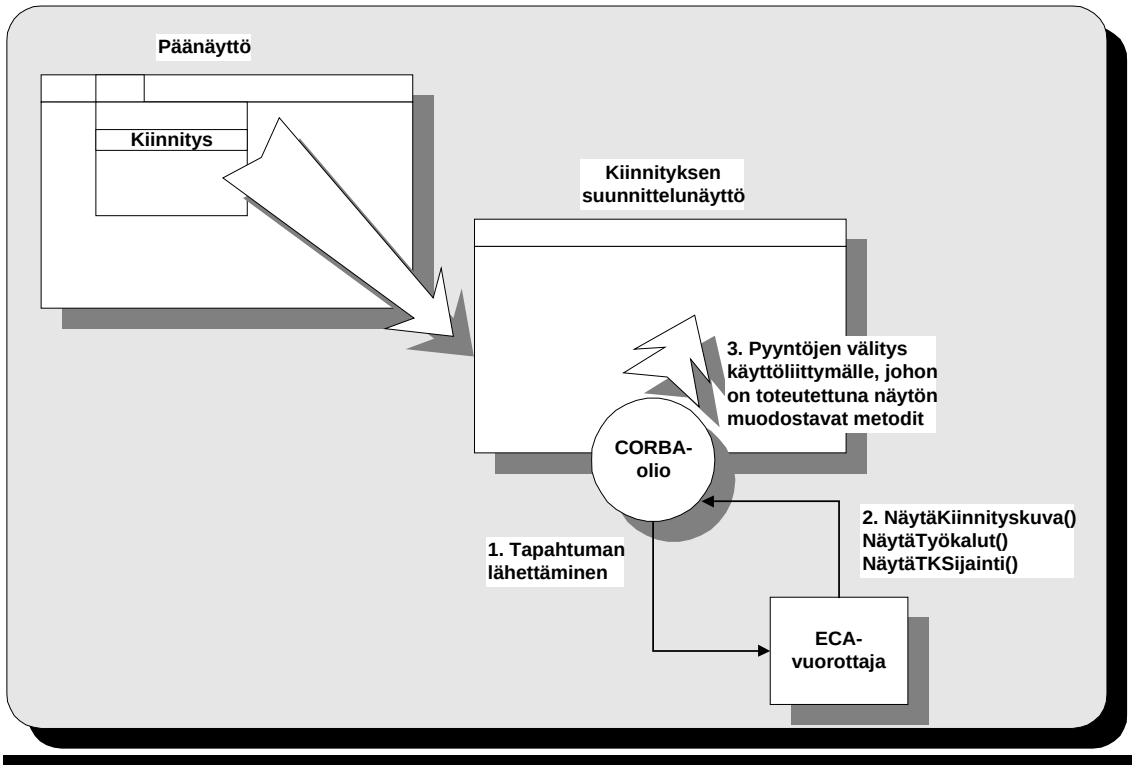
Kuvassa 36 on esitettynä esimerkki valmistusjärjestelmän yhdestä osakokonaisuudesta, työstettävän kappaleen kiinnityksen suunnittelemisesta. Esimerkkiä ei toteutettu pilottijärjestelmässä. Tuoteominaisuuden pakollisuutta kuvaa musta neliö, ja optionaalisia tuoteominaisuuksia voidaan valita rastittamalla kyseinen optio. Järjestelmän toimitukseen kuuluisivat tässä tapauksessa pakollisina itse kiinnityspalvelun lisäksi kiinnityksen suunnittelu ja aktiivisen tilauskannan näyttäminen. Tehty kiinnitys työstetään tässä tapauksessa jollakin koneella, joten järjestelmän koneiden näyttäminen kiinnityksen tekijälle on pakollinen ominaisuus. Optionaalisina tuote-ominaisuuksina on valittu kiinnityksen CAD-kuvan näyttäminen sekä järjestelmän työkalujen seuranta. Työkalut vaativat työkalujen sijaintitiedon näyttämisen ja kiinnityksen tekemisen jälkeen työkalujen tilatietojen päivityksiä. Vastaavat vaatimukset ovat myös kiinnityselementtien seurannalla ja materiaalin hallinnalla. (Relaationuolet on jätetty pois kuvan yksinkertaistamiseksi.) Kuvaustapa tuoteominaisuusmallien visualisointiin on laajennettu MICOM-esitystavasta (Kang & Jang 1994, ks. Kalaoja et al. 1997).



Kuva 36. Esimerkki valmistusjärjestelmän kiinnityspalvelun tuoteominaisuuksista.

Sääntökantaa muodostettaessa voi pakolliset ominaisuudet periatteessa unohtaa, mikäli näiltä ominaisuuksilta ei haluta varioitavuutta. Ensiksi tulee miettiä, mistä tapahtuma saa alkunsa ja mikä tapahtuma on. Esimerkkitapauksessa käyttäjä haluaa tehdä jonkin kappaleen kiinnityksen ja tapahtuman lähettämisen voi yhdistää näytön avautumiseen. Ongelmana on tapahtumien vähäisyys; käyttäjälle tulisi muodostaa näyttö valittujen tuoteominaisuuksien pohjalta. Käytännössä tapahtumia voidaan lähettää vain yksi CORBA-olion muodostimessa. Päivitykset tulee tehdä kiinnityksen jälkeen, joten tarvitaan päivitykset suorittava tapahtuma esimerkiksi näyttöä suljettaessa. Tuoteominaisuusmallien kytkentä ei vaadi sääntökannassa ehto-osaa, koska tuoteominaisuudet ovat joko päällä tai pois.

Oletuskiinnitys ja kiinnityskuva eivät ole sidoksissa muihin tuoteominaisuuksiin. Näille kaikille yksittäisille tuoteominaisuuksille joudutaan tekemään omat säännöt ja omat aliluokat, koska kyseessä on jakamaton ominaisuus. Näiltä osin sääntökannasta muodostuisi mallin kolme (kuva 34) mukainen sillä erolla, että kaikilla säännöillä olisi sama ehto-osa. Jäljelle jäävän osan sääntökannasta muodostavat toisiinsa sidoksissa olevien tuoteominaisuuksien säännöt. Tässä tapauksessa säännöistä muodostuu mallin kaksi (kuva 33) mukaisia sillä erotuksella, että jokaiselle yhdistelmätuoteominaisuudelle muodostuisi oma sääntönsä. Säännöillä on kaksi liipaisinta, aloittava ja lopettava tapahtuma. Nämä tapahtumat on yhdistetty samaan sääntöön, jossa haaraannutaan tapahtumatiedon perusteella oikeaan paikkaan.



Kuva 37. Toiminta näyttöä muodostettaessa. Käyttäjän herätteestä propagoituva toiminnallisuus.

Teoriassa toiminta olisi esitetyn kaltainen (kuva 37). Käytännön toteukseen liittyy epäilemättä useita ongelmia, joista näytön dynaaminen muodostaminen siten, että lopputulos olisi aina hyvä, lienee vaikein. Käytettyyn tuoteominaisuusmallien kuvaustapaan tulisi lisätä tuoteominaisuuteen liittyvä tapahtumatieto, jotta sääntökannan muodostaminen pelkän kuvauksen perusteella helpottuisi. Kun sääntökanta kaikkine johdettuine luokkineen on kerran tehty, se voidaan muodostaa aina sen jälkeen automaattisesti annettujen päälle/pois-tyyppisten tietojen perusteella.

5. Arviointi

Tässä luvussa analysoidaan tehtyä työtä keskittyen tietokantaan toteuteutettujen aktii visuuspiirteiden arviointiin ja vertaamalla kohdassa 0 esitettyyn kehykseen. Järjestelmän jatkokehittämisestä tehdään esityksiä ja arvioita.

5.1 Tulosten analysointia

Kohdassa 0 aktiiviselle tietokannalle esitettyjen olennaisten vaatimusten toteutuminen on koottu taulukkoon 7. Taulukon kohdat vastaavat kohdassa 0 käytettyä numerointia.

Taulukko 7. Yhteenveto toteutetuista ominaisuuksista.

1. Aktiivinen tietokanta toimii myös tavallisena tietokantana.	kyllä
2a. Keinot sääntöjen, ehtojen ja toimintojen määrittelyyn.	ei
2b. Tuki sääntöjen hallintaan ja sääntökannan muutoksille.	kyllä
3a. Tapahtumien havainnointi.	osittain
3b. Ehtojen evaluointi.	osittain
3c. Toimintojen suorittaminen	osittain
3d. Hyvin määritelty suoritussemantiikka.	osittain
3e. Useamman säännön suoritusjärjestys.	kyllä

Kohta 1 on selvä, koska ECA-säännöt toteutettiin tavallisina tietokantaan talletettavina luokkina. Tämä ei periaatteessa estä muita sovelluksia käyttämästä samaa tietokantaa muihin tarkoituksiin.

Kohdan 2 esitys tietokannan liitynnän laajentamisesta on käytännössä mahdoton, mikäli tietokannan valmistaja ei tarjoa mahdollisuutta laajentaa datan määrittelykielen ja tietokannan ominaisuuksia.

Toteutetussa ratkaisussa ECA-säännöt muodostetaan eksplisiittisesti käyttäen tiedon käsittelykieltä, eli C++:aa. Implisiittistä tapahtuman määrittelyä ei tueta, ja havaitut tapahtumat voivat olla pelkästään ulkoisia. Tapahtumat tallentuvat liipaisimen valitsin osaan. Ennen- ja jälkeen-tapahtumia ei ole toteutettu, mikä on tosin mahdollista tiettyyn rajaan asti käyttämällä hyväksi sääntöjen priorisointia. Ehto-osaa ei voida jättää kokonaan pois, ja ehdot eivät ole tietokantaan kohdistuvia kyselyjä. Semanttisesti ehdon poisjättämistä vastaa aina tosi -totuusarvon palauttava ehto-osa. Metodikutsuosissa on mahdollista käyttää tiedonkäsittelykielen ominaisuuksia ja periaatteessa transaktion ohjaamisen käskyjä. Liipaisevaa transaktiota ei voi keskeyttää. Tapahtumatyyppejä on

ainoastaan yksi, ulkoinen primitiivinen tapahtuma. Vaatimus 2a ei toteudu. Tapahtumia, ehtoja ja toimintoja ei voi määritellä tietokannan käyttämällä tiedon määrittelykielellä.

Sääntökanta on muutettavissa käyttäen tiedon käsittelykieltä, ja säännöt sijaitsevat tietokannassa muiden luokkien tapaan. Uusia sääntöjä voidaan määritellä ja vanhoja poistaa. Tämä tosin vaatii tähän tarkoitukseen tehdyn ohjelman toteuttamisen. Säännöt ovat kytkettävissä päälle tai pois, ja olemassa olevia sääntöjä voidaan selata esimerkiksi tietokannan mukana tulevalla selaimella. Vaatimuksen 2b voi katsoa olevan voimassa.

Järjestelmällä on suoritusmalli, joskin yksinkertainen, joten vaatimus 3 täyttyy. Sovellusohjelmoija on täysin vastuussa tapahtumien signaloinnista, joten vaatimus 3a toteutuu ainoastaan tapahtuman havaitsemisen osalta. Järjestelmä pystyy ehdon evaluointiin, mutta tietokantaan tehtäviä kyselyitä ei toteutettu. Vaatimus 3b toteutuu osittain. Toimintoja pystytään suorittamaan ja tapahtumatietoa välitetään ehto-osasta toiminto-osalle. Järjestelmässä ei voi hallita CORBA-olioiden transaktioita, joten 3c toteutuu osittaisesti. Yhdistelmätapahtumia ei ole toteutettu, eikä niihin liittyen tapahtumahistoriaa. Vaihtoehtoja tapahtumien käsittelylle ei ole ja kytkentämalli on pelkästään siirretty. Tapahtumat ovat aina instanssikohtaisia. CORBA-oliot eivät muuta tietokannan tilaa millään tavalla, joten tietokannan tila, joka näkyy ehdon evaluoinnille, on aina senhetkinen. Vaatimus 3d toteutuu ainoastaan kykentämallin osalta. Sääntöjen suoritusjärjestys on käyttäjän määriteltävissä, joten vaatimus 3e täyttyy.

Koska käytettiin yhtä käyttöjärjestelmää, CORBA-arkkitehtuurin mahdollistamalle hetero-geenisyydelle ei ollut tarvetta pilottijärjestelmässä. Järjestelmää voidaan kuitenkin teoriassa käyttää heterogeenisenä järjestelmää ohjaavana ratkaisuna. Mikäli järjestelmää käytettäisiin pelkästään tuoteominaisuuksien aktivointiin, Dittrichin (et al. 1995) esittämä toisen tason aktiivinen tietokanta olisi riittävä.

Kolmannen tason aktiiviselle tietokannalle asetetut vaatimukset (taulukko 4) toteutuvat ainoastaan osittain. Näin tulkiten pilottiohjelmistossa käytettyä tietokantaa ei voi pitää aktiivisena. Järjestelmässä ei kuitenkaan tehdä periodisesti suoritettavia kyselyitä järjestelmän tilasta. Myöskään järjestelmän olioissa ei ole suoraan tietokantaan sidoksissa olevaa koodia. Tietokanta reagoi ainoastaan sille lähetettyihin tapahtumiin ja suorittaa toiminnot sen mukaisesti. Kun järjestelmässä ei esiinny tapahtumia, ECA-vuorottaja ‘odottaa’ passiivisesti vapauttaen koneen kaikki resurssit muuhun käyttöön. Näihin seikkoihin perustuen järjestelmän tietokanta on aktiivinen.

Dittrichin et al. (1995) esittämiä vaatimuksia voi pitää perusteltuina pyrittäessä rakentamaan yleiskäyttöistä ja luotettavasti toimivaa aktiivista tietokantaa. Esitettyjä vaatimuksia voisi kuitenkin tarkentaa ajan mukaisesti juuri hajautuksen osalta, CORBA-palvelut huomioiden. Esitystä tietokannan liitynnän laajentamisesta sääntöjen määrittelyyn voisi myös tarkentaa. Lopputulos on käyttäjän kannalta sama, vaikka sääntöjen muodostamisessa käytettäisiinkin tiedonkäsittelykieltä.

5.2 Järjestelmän kehittäminen

ECA-vuorottajaa käytettiin aluksi ilman minkäänlaista ehdon evaluointia sääntöjen ehto-osien palauttaessa aina arvon tosi. Tämä johtui vuorottajan käytöstä pelkkään tapahtumien välittämiseen järjestelmän eri olioiden välillä. Vuorottajan käyttö pelkästään tapahtumien välittämiseen ei tuo lisäarvoa järjestelmälle. Mikäli järjestelmää käytettäisiin pelkästään tuoteominaisuuksien hallintaan, sääntöketjut ilman ehto-osia riittäisivät tähän tarkoitukseen.

ECA-vuorottajalla ohjattava, ORB-väylässä hajautettu järjestelmä laajenee helposti pelkästään instantioimalla uusia CORBA-oliota järjestelmään, mutta toiminnallisuuden laajentaminen tai laventaminen vaatii muutoksia sääntökantaan. Mikäli kyseessä on kokonaan uusi toiminta, tässä ratkaisumallissa joudutaan johtamaan uusia aliluokkia sääntökantaan ja toteuttamaan lopullinen toiminallisuus itse CORBA-olioon tehtävin lisäyksin, esimerkiksi lisäämällä olion liityntään uusi metodi ja tekemällä sille toteutus. Jotta vältytään CORBA-olioiden liityntöjen paisumiselta, tulee toiminnallisuuden muuttaminen suunnitella mahdollisimman pitkälle parametroimalla sopivasti ECA-vuorottajan kutsumia metodeja. Tämä ei kuitenkaan poista aliluokittamisen tarvetta sääntö-kannassa, mikäli kaikkea toiminnallisuutta ei haluta toteuttaa yhdessä metodikutsuluokassa.

ECA-vuorottajan suorituskykyä tulee parantaa. Tämä on tärkeä seikka, koska käytännössä yhden tapahtuman käsittelyyn käytetty aika paljolti määrää minimitapahtumavälin. Mikäli tapahtumia ECA-vuorottajalle lähetetään nopeammin kuin se pystyy käsittelemään, pitkällä aikavälillä puskurit todennäköisesti täyttyvät ja tapahtumia menetetään järjestelmän ennen pitkää kaatuessa. Suorituskykyä voi parantaa määrittelemällä omat muistinvaraiset tietorakenteet sääntöjen käsittelyyn ja lukemalla sääntökanta muistiin sekä välttämällä raskasta levylle kirjoitusta sääntöä evaluoitaessa. Tässä joudutaan kuitenkin kompromisseihin, mikäli esimerkiksi halutaan toteuttaa suoritettujen tapahtumien auditointia myöhemmin tai mahdollisesti palauttaa järjestelmän kaatumisen aiheuttanutta tapahtumaa edeltänyt tila.

Dynaamisten metodikutsujen (DII) käytön vaikutus vuorottajan suorituskykyyn tulisi mitata. Optimistisesti arvioiden on mahdollista päästä muistinvaraisella vuoronnuksella 40 millisekunnin suoritusaikoihin tapahtuman käsittelyssä. Tällä hetkellä suorituskyky on n. 70 - 550 millisekuntia säännöstä riippuen. Viive on pienin silloin, kun lähetetylle tapahtumalle ei ole määritelty sääntöä. Tällöin liipaisimien koontiluokasta ainoastaan haetaan liipaisimia, mutta yhtään tapahtumaan liittyvää liipaisinta ei löydy. Mikäli kuitenkin käytetään staattista metodin kutsumista, tulee CORBA-olioiden liitynnät suunnitella huolellisesti käyttäen apuna abstrakteja kantaluokkia, jotta vuorottajaan sidoksissa olevien liityntöjen määrä saadaan pidettyä mahdollisimman pienenä. Näin vuorottaja saadaan vähemmän sidonnaiseksi sitä ympäröivästä ORB-maailmasta.

Sääntökannan muodostaminen ohjelmallisesti on hyvin työlästä ja virhealtista. Tuotantokäytössä ECA-sääntöeditori on lähes välttämätön. Hyvän ECA-editorin tekeminen on vaativa tehtävä. Säännöt tulisi jollakin tavalla pystyä visualisoimaan ja yksinkertainen sääntökannan oikeellisuuden evaluointi tulisi pystyä suorittamaan. Sääntökantaa ja sääntöeditoria joudutaan kehittämään

rinnakkain, sillä sääntökantaan tehdyt muutokset heijastuvat editoriin järjestelmän perustuessa käännettyyn koodiin, mikäli editorilla halutaan instantioida sääntökantaan lisättyjä luokkia.

Tarkkoja vasteaikoja ECA-vuorottajan käytölle on parhaimmassakin tapauksessa vaikea määritellä. Asiaan vaikuttavat useat seikat, kuten tapahtuman välitykseen käytetyn median kuormitus, ECA-vuorottajan kuormitus sekä sääntöön muodostuvien kaksisuuntaisten kutsujen määrä. Mikäli kaksisuuntaiset kutsut kohdistuvat raskaasti kuormitettuun järjestelmän komponenttiin, säännön suoritus hidastuu tältä osin ECA-vuorottajan odottaessa paluuarvoa. Vuorottaja tulisikin säikeistää. Näin sääntöjen suoritukseen saataisiin lisää rinnakkaisuutta. Epädeterministisyys rajoittaa konseptin käyttöä kovissa reaaliaikajärjestelmissä.

CORBA-olioiden pysyvyys on erittäin tärkeä seikka koko järjestelmän toimintaa ajatellen. Tämän asian toteuttamiseen tulisi kiinnittää suurta huomiota kehitettäessä järjestelmää edelleen. Ohjattavien komponenttien tilan tulee näkyä tietokannassa. Tällöin ehto-osan evaluoinnissa ei tarvitse kuormittaa ORB-väylää ja koko järjestelmän tila olisi tarvittaessa palautettavissa tietokannasta. Yksisuuntaisen kutsun käyttämiseen liittyvät ongelmat tulisi ratkaista, jotta varmistuttaisiin tiedon oikeellisuudesta ja operaatioiden jakamattomuudesta.

Järjestelmän toimintaa ei voi pitää kovin vakaana ja ongelmia esiintyi tapahtumien nopeassa perättäisessä lähettämisessä sekä käytettäessä yhtä aikaa tavallista tietokantaa käyttävää sovellusta ja tapahtuman lähettävää sovellusta. Lisätyötä lopullisen tuotteen aikaansaamiseksi voi pitää suurena.

6. Yhteenveto

Tässä tutkielmassa etsitään vastauksia ohjelmiston joustavuusominaisuuksien käyttömahdollisuuksiin hajautetussa tehtäväkriittisessä järjestelmässä. Tutkielman keskeisimmät aihealueet ovat aktiiviset tietokannat ja hajautuksen mahdollistavan CORBA-arkkitehtuurin yhdistäminen aktivoituun tietokantaan. Tutkielmassa käsitellään aktiivisia tietokantoja ja niille esitettyjä vaatimuksia soveltavasta näkökulmasta. Tehtyä työtä verrataan esitettyihin vaatimuksiin. ECA-sääntöjen muodostamiseen esitetään erilaisia vaihtoehtoja ja esimerkkinä tuoteominaisuusmallin kuvaamisesta sääntöinä esitetään yksi skenaario. Seuraavassa ovat vastaukset tutkimusongelmiin.

Mitä ohjelmiston joustavuusmekanismien käytöllä saavutetaan verrattuna komponenttipohjaiseen ohjelmistoarkkitehtuuriin?

ECA-konseptin käyttäminen mahdollistaa ohjelmiston komponenttien löysät kytkennät ja järjestelmän ominaisuuksien joustavan muuttamisen. Ohjattavista komponenteista muodostuu kuitenkin käytetyllä tekniikalla perinteisiä monoliittisia sovelluksia, joihin muutettavat tuoteominaisuudet ovat valmiiksi toteutettuina ECA vuorottajan toimiessa pelkästään kytkimenä. Pilottiohjelmistossa komponenttien löysää kytkentää ei saavutettu, koska parametritieto puuttui tapahtumatiedosta.

CORBA-arkkitehtuurin mahdollistama joustava laajennettavuus tuo selvää lisäarvoa järjestelmälle, mutta järjestelmän hajauttaminen itsenäisesti toimiviin olioihin aiheuttaa ongelmia. Lukkiutumisongelmat johtavat ratkaisuun, jossa vuorottajan tarvitseman tilatiedon oikeellisuudesta ei voi olla varma. Lisäksi ECA-konseptin vaatima tilatiedon lukeminen tietokannasta ilman toimivaa tietokantasovitinta muodostui ongelmaksi.

Miten joustavuusominaisuudet tulee huomioida tehtäväkriittisessä hajautetussa järjestelmässä?

Joustavuuden saavuttamiseksi tulee tuntea ohjattava järjestelmä ja sen varioitavat tuoteominaisuudet. ECA-sääntöjen muodostamiseen ei voi esittää yhtä ainoaa oikeaa vaihtoehtoa, vaan säännöt muodostuvat ohjattavan toiminnan mukaan. ECA-konseptin staattinen CORBA-toteutus rajoittaa joustavuutta. Koska järjestelmä perustuu käännettyyn koodin ja vahvaan tyypintarkistukseen, ECA-mallissa joudutaan johdettujen luokkien käyttöön. Mikäli esimerkiksi järjestelmään tuodaan kokonaan uusi olio, tätä ei voida suoraan käyttää ECA-säännön metodikutsuissa tai ehdoissa, koska yhteydenmuodostamiseen tarvittavaa tietoa (otsikkotiedostoa) ei ole ollut tietokantaan talletettavia luokkia tehtäessä. Tätä rajoitusta voi kiertää kutsuttavien olioiden onnistuneella luokkahierarkialla, jolloin yhteys muodostetaan aina kantaluokkaan. Näin on mahdollista saavuttaa yleiskäyttöisyyttä, joka rajoittuu kantaluokan tarjoaman liitynnän käyttöön.

Aktiivinen tietokanta voidaan toteuttaa vahvasti tyypitettyä käännettävää kieltä käyttäen. Tarve yleiskäyttöisille metodikutsu- ja ehtoluokille on kuitenkin ilmeinen, jotta vältyttäisiin suurilta määriltä johdettuja luokkia. Tulisikin siis tutkia CORBA-arkkitehtuurin tarjoama dynaaminen

metodien kutsuminen (DII) ja pyrkiä sen pohjalta muodostamaan mahdollisimman yleiskäyttöiset luokat. Samoin olisi olennaista saada aikaan parametrien välittäminen tapahtuman mukana, jotta järjestelmän olioiden väliset kytkennät saataisiin mahdollisimman vähäisiksi. Parametrien välittämisessä joudutaan todennäköisesti jonkinlaisiin kompromisseihin yleiskäyttöisyyden ja läpinäkyvyyden välillä.

Laajentaminen vaatimusten mukaiseksi aktiiviseksi tietokannaksi olisi epäilemättä suuri ponnistus. Pyrkimys yleiskäyttöisyyteen ja tyypittömyyten vahvasti tyypitetyllä käännetyllä ohjelmointikielellä on verrattavissa jatkuvaan vastavirtaan uimiseen ja johtaa kompromissiratkaisuihin. Molemmat käytetyistä toteuttavista teknologioista ovat tuotteina käytettävissä erikseen ja täyttävät tehtävänsä. Sen sijaan niiden yhtäaikainen käyttö aiheutti ongelmia, joiden syitä voi vain arvailla. Tuotteena käytetyn tietokannan arvosanaa laskee ORB-sovittimen puuttuminen Windows NT -versiosta. Valmistaja toimitti sovittimen Sun Solaris -käyttöjärjestelmälle tarkoitetun version lähdekoodin, jota ei kuitenkaan saatu sovitettua NT-ympäristöön. IONAn Orbix ORB -toteutuksen huonoimpana puolena on korkeintaan välttävänä pidettävä dokumentointi.

Aktiivisten tietokantojen yleistymistä hajautetuissa ohjausjärjestelmissä hidastavat käytettyjen teknologioiden uutuus ja teknologioiden yhdistämiseen liittyvät ongelmat. Joustavuusmekanismien käytöllä saavutettavat edut ovat kuitenkin niin selvät, että on vain ajan kysymys, kun näemme niitä tuotantokäytössä. Siirtymäkaudesta uusien mukautuvien järjestelmien käyttöön muodostunee kuitenkin pitkä.

Lähteet

Baker, S. 1997. CORBA Distributed Objects Using Orbix. Harlow: Addison-Wesley. ISBN 0-201-92475-7.

Barry, D. K. 1996. The object database handbook: how to select, implement, and use object-oriented databases. New York: John Wiley & Sons. ISBN 0-471-14718-4.

Bernstein, P. A. 1996. Middleware: A Model for Distributed System Services. Communications of the ACM, Vol. 39, No. 2, s. 86 - 98.

Cattell, R. G. G. & Barry, D. K. (eds.). 1997. Object Database Standard: ODMG 2.0. San Fransisco, California: Morgan Kaufman Publishers. ISBN 1-55860-463-4.

Chakravarthy, S. 1995. Early Active Database Efforts: A Capsule Summary. IEEE Transactions on Knowledge and Data Engineering, Vol. 7, No. 6., s. 1008 - 1010.

Chakravarthy, S. 1997. Sentinel: An Object-Oriented DBMS With Event Based Rules. In: Peckman, J. M. (ed.). Proceedings of ACM SIGMOD International Conference on Management of Data. SIGMOD Record, Vol. 26, Issue 2, s. 572 - 575.

Chakravarthy, S., Anwar, E., Maugis, L. & Mishra, D. 1994. Design of Sentinel: an object-oriented DBMS with event-based rules. Information and Software Technology, Vol. 36, No. 9., s. 555 - 568.

Dayal, U., Buchmann, A. P. & McCarthy, D. R. 1988. Rules Are Objects Too: A Knowledge Model For An Active, Object-Oriented Database System. In: Dittrich, K. R. (ed.). Advances in Object-Oriented Database Systems. 2nd International Workshop on Object-Oriented Database Systems, LNCS 334. Berlin: Springer-Verlag. S. 129 - 143.

Dellarocas, C. 1997. The SYNTHESIS Environment for Component-Based Software Development. In: Budgen, D., Hoffnagle, G. & Trienekens, J. (eds.). Proceedings of the Eighth IEEE International Workshop on Software Technology and Engineering Practice Incorporating Computer Aided Software Engineering. Los Alamitos, California: IEEE Computer Society. S. 434 - 443. ISBN 0-8186-7840-2.

Deri, L. 1997. Yasmin: a Component Based Architechture for Software Applications. In: Budgen, D., Hoffnagle, G. & Trienekens, J. (eds.). Proceedings of the Eighth IEEE International Workshop on Software Technology and Engineering Practice Incorporating Computer Aided Software Engineering. Los Alamitos, California: IEEE Computer Society. S. 4 - 12. ISBN 0-8186-7840-2.

Dittrich, K. R., Gatziu, S. & Geppert, A. 1995. The Active Database Management System Manifes to: A Rulebase of ADBMS Features. In: Sellis, T. (ed.). Rules in Database Systems: Second Inter national Workshop Proceedings, RIDS ‘95, LNCS 985. Berlin: Springer. S. 3 - 17. ISBN 3-540-60365-4.

Dittrich, K. R., Kotz, A. M. & Mülle, J. A. 1986. An Event/Trigger Mechanism to Enforce Complex Consistency Constraints in Design Databases. SIGMOD RECORD, Vol. 15, No. 3, s. 22 - 36.

Elmasri, R. & Navathe, S. B. 1994. Fundamentals of database systems. 2nd ed. Redwood City, California: The Benjamin/Cummings Publishing Company. ISBN 0-8053-1748-1.

Gehani, N. H., Jagadish, H. V. & Shmueli, O. 1992. Event Specification in an Active Object-Oriented Database. In: Stonebraker, M. (ed.). Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data. SIGMOD Record, Vol. 21, Issue 2, s. 81 - 90.

Greenwood N. R. 1986. FMS-control and communications. The problems and the potential. In: Rathmill, K. (ed.). Proceedings of the 5th International Conference on Flexible Manufacturing Systems. Kempston, Bedford: IFS Publications. S. 1 - 7. ISBN 0-948507-17-9.

Hanson, E. N. 1996. The Design and Implementation of the Ariel Active Database Rule System. IEEE Transactions on Knowledge and Data Engineering, Vol. 8, No. 1, s. 157 - 172.

Hartley, J. 1984. FMS at work. Bedford: IFS Publications. ISBN 0-903608-62-6.

Huemer, C., Kappel, G. & Vieweg, S. 1995. Migration in Object-oriented Database Systems—A Practical Approach. Software—Practice And Experience, Vol. 25, No. 10, s. 1065 - 1096.

Ihme, T. & Niemelä, E. 1996. Adaptability in Object-Oriented Embedded and Distributed Software. In: Mühlhäuser, M. (ed.). Special issues in object oriented programming: workshop reader of the 10th European Conference on Object Oriented Programming ECOOP ‘96. Heidelberg: dpunkt, Verl. für digitale Technologie. S. 29 - 36. ISBN 3-920993-67-5.

Järvinen, P. & Järvinen, A. 1995. Tutkimustyön metodeista. Tampere: Opinpaja Oy. ISBN 951-97113-1-7.

Kalaoja, J., Niemelä, E. & Perunka H. 1997. Feature Modelling of Component-Based Embedded Software. In: Budgen, D., Hoffnagle, G. & Trienekens, J. (eds.). Proceedings of the Eighth IEEE International Workshop on Software Technology and Engineering Practice Incorporating Computer Aided Software Engineering. Los Alamitos, California: IEEE Computer Society. S. 444 - 451. ISBN 0-8186-7840-2.

Kappel, G., Rausch-Schott, S., Retschitzegger, W. & Vieweg, S. 1994. TriGS: Making a passive object-oriented database active. Journal of Object-Oriented Programming, Vol. 7, No. 4., s. 40 - 51, 63.

King, S. S. 1992. Middleware!. Data Communications, Vol. 21, No. 4., s. 58 - 67.

Loftus, C. W., Sherratt, E. M., Gautier, R. J., Grandi, P. A. M., Price, D. E. & Tedd, M. D. 1995. Distributed Software Engineering. London: Prentice Hall. ISBN 0-13-342676-9.

Loomis, M. E. S. 1995. Object Databases: the Essentials. Reading, Massachusetts: Addison-Wesley. ISBN 0-201-56341-X.

Machura, M. 1996. Managing Information in a Co-operative Object Database System. Software—Practice And Experience, Vol. 26, No. 5, s. 545 - 579.

McCarthy, D. R. & Dayal, U. 1989. The Architechture of an Active Data Base Management System. In: Clifford, J., Lindsay, B. & Maier, D. (eds.). Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data. SIGMOD RECORD, Vol. 18, No. 2, s. 215 - 224.

Mortimer, J. (ed.). 1982. The FMS Report: Ingersoll Engineers. Bedford: IFS Publications.
ISBN 0-903608-31-6.

Mowbray, T. & Zahavi, R. 1995. The Essential CORBA: Systems Integration Using Distributed Objects. New York: John Wiley & Sons. ISBN 0-471-10611-9.

Mowbray, T. J. & Malveau, R. C. 1997. Corba Design Patterns. New York: John Wiley & Sons. ISBN 0-471-15882-8.

Orfali, R., Harkey, D. & Edwards, J. 1997. Instant CORBA. New York: John Wiley & Sons. ISBN 0-471-18333-4.

Päivike, H. 1991. Software Design for Embedded Systems with Hard Real-Time Constraints. Licentiate thesis. Oulu: University of Oulu, Department of Electrical Engineering.

The Common Object Request Broker: Architecture and Specification. July 1995. Object Management Group, Vol. I, Revision 2.0.

Tran, V., Liu, D. & Hummel, B. 1997. Component-based Systems Development: Challenges and Lessons Learned. In: Budgen, D., Hoffnagle, G. & Trienekens, J. (eds.). Proceedings of the Eighth IEEE International Workshop on Software Technology and Engineering Practice Incorporating Computer Aided Software Engineering. Los Alamitos, California: IEEE Computer Society. S. 452 - 462. ISBN 0-8186-7840-2.

Widom, J. & Ceri, S. 1996. Introduction to Active Database Systems. In: Widom, J. & Ceri, S. (eds.). Active database systems: triggers and rules for advanced database processing. San Fransisco, California: Morgan Kaufmann Publishers. S. 1 - 41. ISBN 1-55860-304-2.

Widom, J. 1996. The Starburst Active Database Rule System. IEEE Transactions on Knowledge and Data Engineering, Vol. 8, No. 4, s. 583 - 595.

Woolridge, M. & Jennings, N. R. 1995. Agent Theories, Architechtures, and Languages: A Survey. In: Woolridge M. J. & Jennings N. R. (eds.). Intelligent Agents, ECAI-94 Workshop on Agent Theories, Architechtures, and Languages., LNAI 890. Berlin: Springer-Verlag. S. 1 - 39. ISBN 3-540-58855-8.